# Случайные процессы. Прикладной поток.

## Практическое задание 4

**Правила:**

- Выполненную работу нужно отправить на почту `probability.diht@yandex.ru`, указав тему письма `"[СП17] Фамилия Имя - Задание 4"`. Квадратные скобки обязательны. Вместо `Фамилия Имя` нужно подставить свои фамилию и имя.
- Прислать нужно ноутбук и его pdf-версию. Названия файлов должны быть такими: `4.N.ipynb` и `4.N.pdf`, где `N` - ваш номер из таблицы с оценками.
- При проверке никакой код запускаться не будет.

---

Для выполнения задания потребуются следующие библиотеки: hmmlearn, librosa. Следующими командами можно их поставить (Ubuntu):

sudo pip3 install hmmlearn

sudo pip3 install librosa

---

### 1. Скрытые марковские модели (2 балла)

Реализация методов является полезной, но технически сложной, поэтому мы воспользуемся готовой реализацией `hmmlearn`. Документация http://hmmlearn.readthedocs.io/ (http://hmmlearn.readthedocs.io/). Интерфейс данной библиотеки максимально близок к библиотеке `scikit-learn`.

Все необходимые комментарии по интерфейсу библиотеки `hmmlearn` приведены в коде далее. Следуйте указаниям.

In [1]:
```
import numpy as np
import sys
from hmmlearn import hmm

import matplotlib.pyplot as plt
%matplotlib inline
```

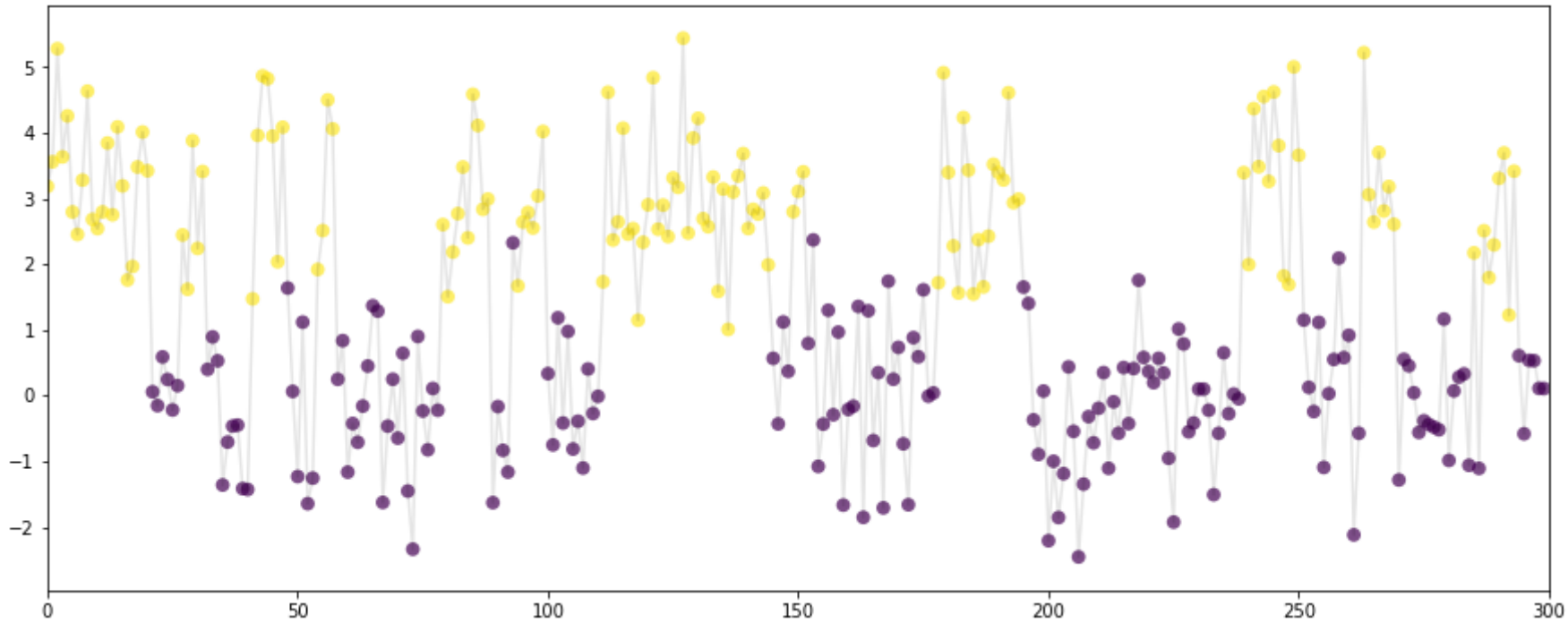
In [6]:
```
# Если при использовании библиотеки будут появлятся различные warnings,
# раскомментируйте и выполните следующий код
import warnings
warnings.filterwarnings("ignore")
```

Зададим некоторую скрытую марковскую модель

In [179]:
```
# Объявление скрытой марковской модели с двумя скрытыми состояниями,
# в которой предполагается, каждое состояние может генерировать
# гауссовский случайный вектор с произвольной матрицей ковариаций.
# Используется метод Витерби.
# Поставьте 'map', чтобы использовать метод forward-backward.
model = hmm.GaussianHMM(n_components=2, covariance_type='full',
                        algorithm='viterbi')

# Параметры марковской цепи - начальное состояние и матрица переходных вероятностей

model.startprob_ = np.array([0.6, 0.4])
model.transmat_ = np.array([[0.9, 0.1],
                            [0.07, 0.93]])
# слишком хорошая матрица, большие диагональные значения.

# Параметры условных распределений Y_j при условии X_j - вектора средних и
# матрица ковариаций по количеству состояний. Поскольку в данном случае
# распределения одномерные, ниже записаны два вектора размерности 1
# и две матрицы размерности 1x1
model.means_ = np.array([[0.0], [3.0]])
model.covars_ = np.array([[[1]], [[1]]])
```

Сгенерируем некоторую последовательность с помощью определенной выше модели.

In [180]:
```
size = 300
Y, X = model.sample(size)  # Y наблюдаемы, X скрытые

plt.figure(figsize=(15, 6))
plt.plot(np.arange(size), Y[:, 0], color='black', alpha=0.1)
plt.scatter(np.arange(size), Y[:, 0], c=np.array(X), lw=0, s=60, alpha=0.7)
plt.xlim((0, size))
plt.show()
```



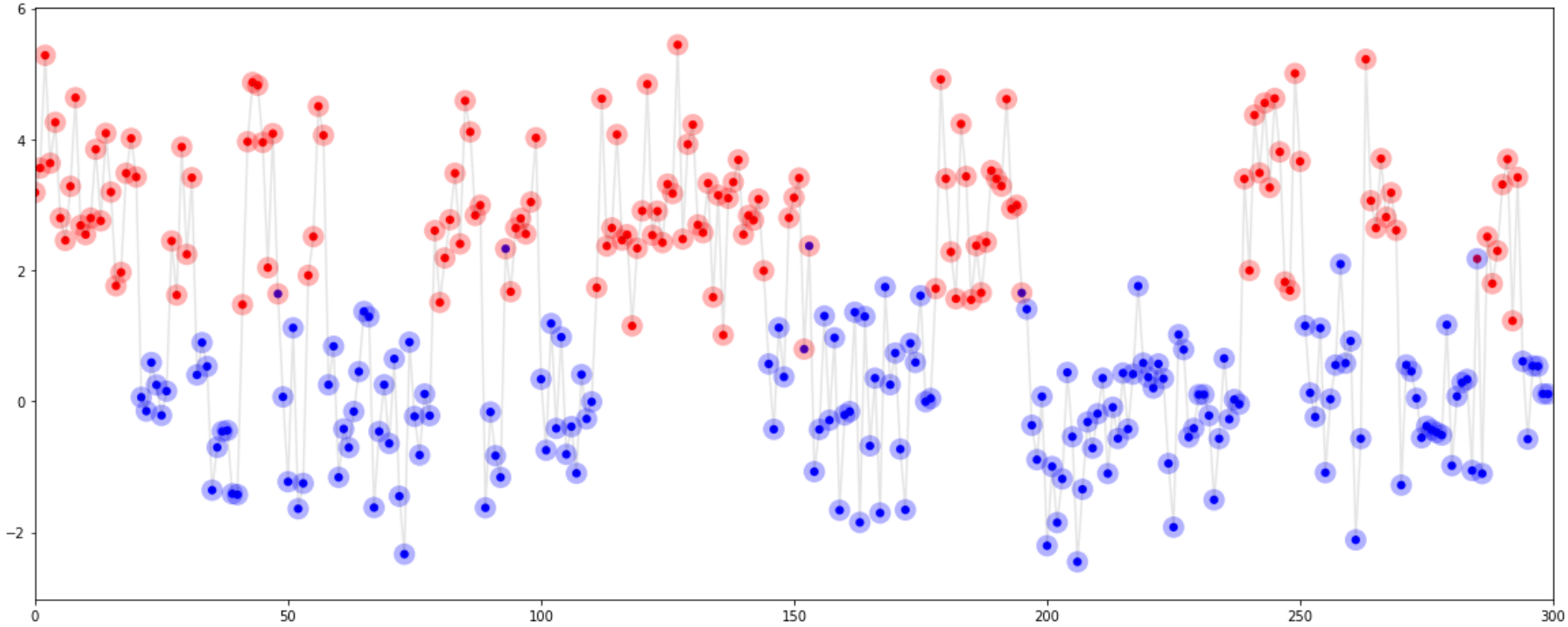

На основе сгенерированной выше последовательности оценим параметры ("обучим") скрытой марковской модели и значения скрытых состояний.

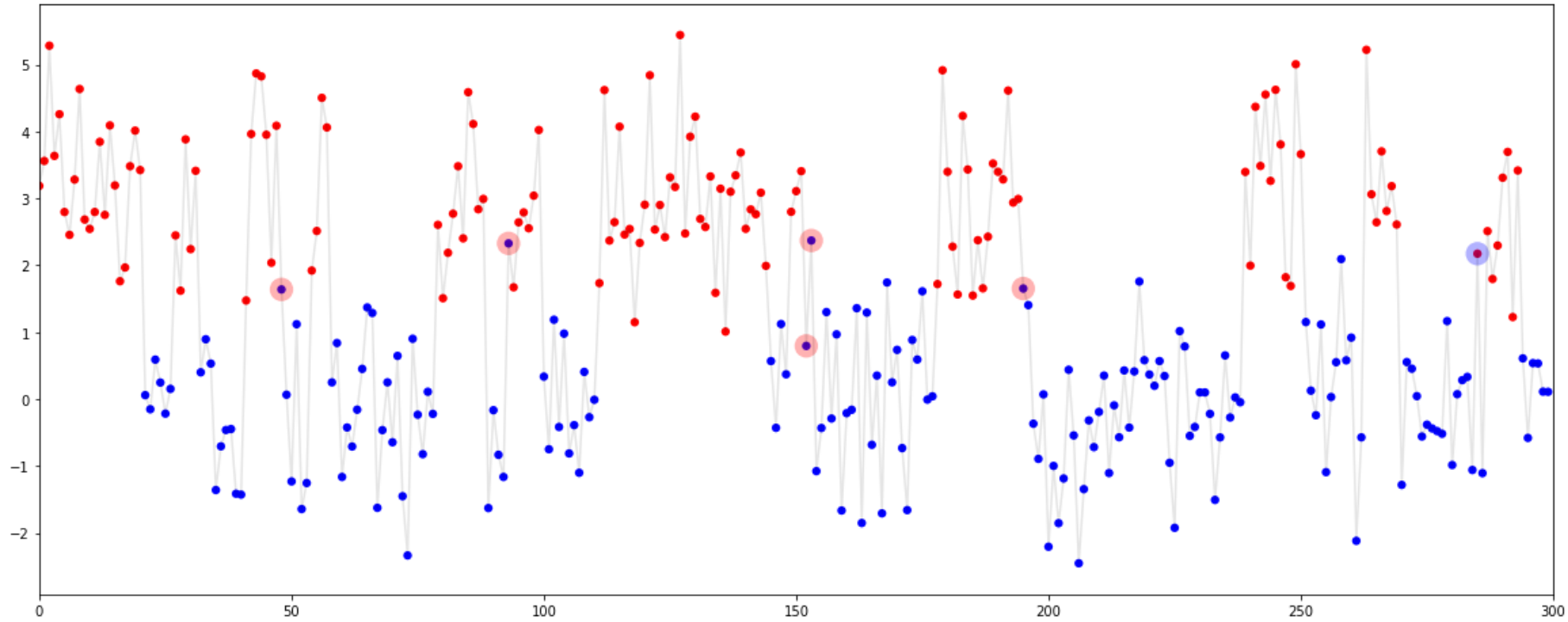
In [181]:
```
# Объявление скрытой марковской модели, в которой при оценке параметров
# будет производиться не более n_iter итераций EM-алгоритма.
remodel = hmm.GaussianHMM(n_components=2, covariance_type="full",
                          n_iter=100, algorithm='viterbi')

# Оценка параметров ("обучение")
remodel.fit(Y)

# Оценка ("предсказание") значений скрытых состояний
X_predicted = remodel.predict(Y)
```

Теперь изобразим полученные результаты. На обоих графиках непрозразными маленькими кружочками отмечена исходная последовательность. Полупрозрачными большими кружочками отмечены оценки значений скрытых состояний. На первом графике отмечены все такие точки, на втором только те из них, оценка значения скрытого состояния получилась непрпавильно.

In [182]:
```
colors = np.array(['blue', 'red'])

# Состояния определяются с точностью до их перестановки.
# При необходимости меняем местами состояния
if (X != X_predicted).sum() > size / 2:
    X_predicted = 1 - X_predicted

print("Accuracy = ", (X == X_predicted).sum() / len(X))

plt.figure(figsize=(20, 8))
plt.plot(np.arange(size), Y[:, 0], color='black', alpha=0.1)
plt.scatter(np.arange(size), Y[:, 0], c=colors[np.array(X)],
            lw=0, s=40, alpha=1)
plt.scatter(np.arange(size), Y[:, 0], c=colors[np.array(X_predicted)],
            lw=0, s=250, alpha=0.3)
plt.xlim((0, size))
plt.show()

plt.figure(figsize=(20, 8))
plt.plot(np.arange(size), Y[:, 0], color='black', alpha=0.1)
plt.scatter(np.arange(size), Y[:, 0], c=colors[np.array(X)],
            lw=0, s=40, alpha=1)
plt.scatter(np.arange(size)[X != X_predicted], Y[:, 0][X != X_predicted],
            c=colors[np.array(X_predicted)[X != X_predicted]],
            lw=0, s=300, alpha=0.3)
plt.xlim((0, size))
plt.show()
```

```
Accuracy =  0.98
```


6
4
2
0
−2
0
50
100
150
200
250
300

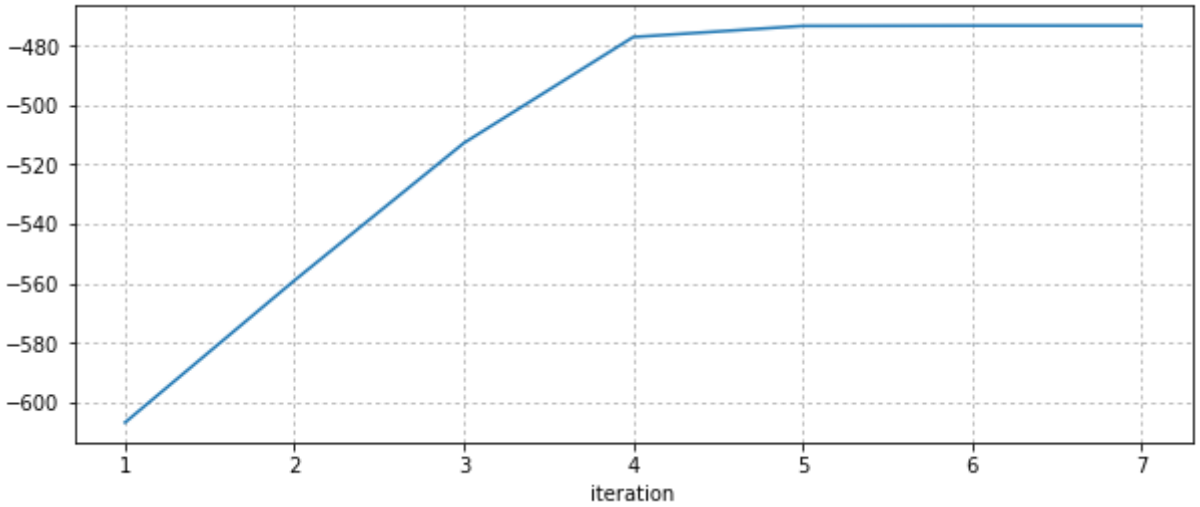



Как понять, что EM-алгоритм сошелся? Для этого нужно посчитать значение некоторого функционала (см. презентацию), который умеет считать библиотека `hmmlearn`, поэтому мы всего лишь посмотрим на его значения. Данная функциональность в библиотеке реализованна слишком странно. Следуйте комментариям.

In [11]:
```
saved_strerr = sys.stderr  # сохраним в переменную поток вывода ошибок
sys.stderr = open('est_values.txt', 'w')  # и перенаправим его в файл

# ============
# Для вывода значений функционала нужно поставить параметр verbose
remodel = hmm.GaussianHMM(n_components=2, covariance_type="full",
                          n_iter=100, verbose=True)
remodel.fit(Y)
X_predicted = remodel.predict(Y)
# ============

# Возвращаем все, как было
sys.stderr = saved_strerr
```

Теперь можно загрузить значения и построить график

In [12]:
```
values = np.loadtxt('./est_values.txt')

plt.figure(figsize=(10, 4))
plt.plot(values[:, 0], values[:, 1])
plt.xlabel('iteration')
plt.grid(ls=':')
plt.show()
```




Выполните те же операции для следующих двух случаев

- скрытая марковская цепь имеет три скрытых состояния;
- распределение $Y_j$ при условии $X_j$ является двумерным гауссовским.

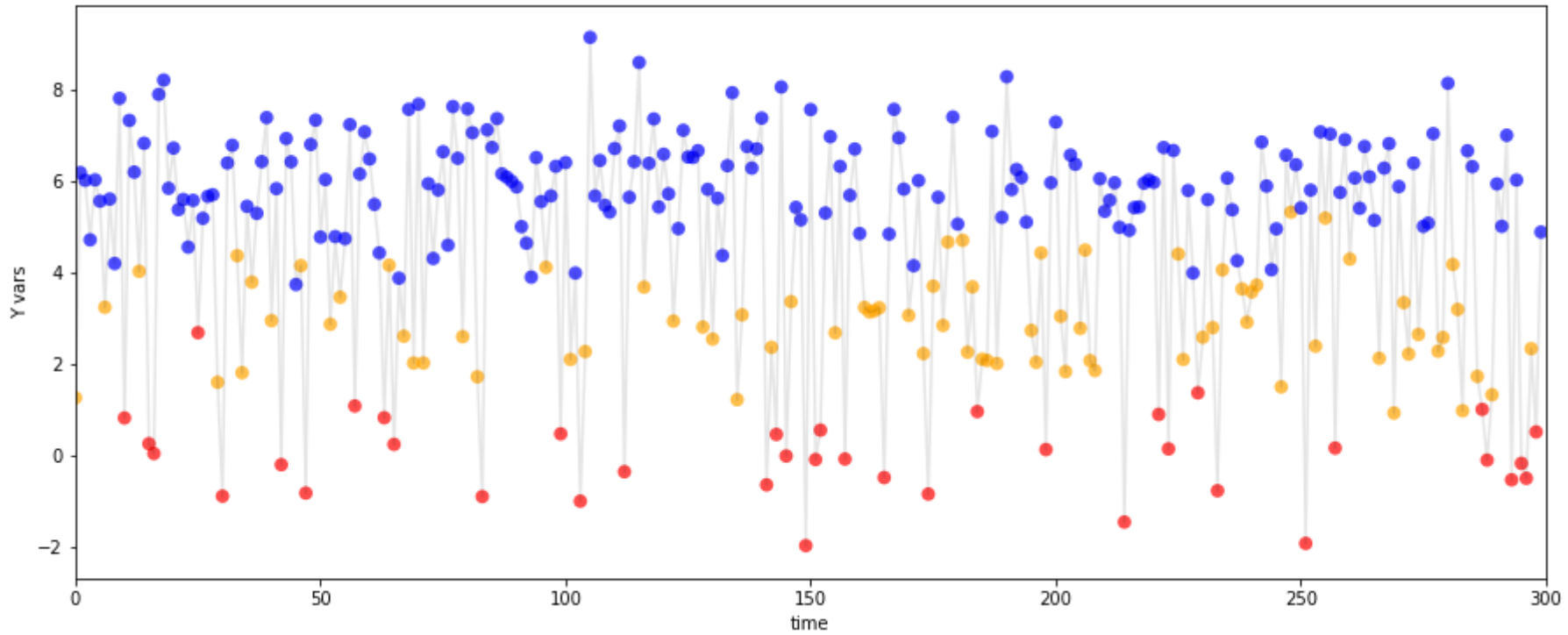
## 1.1 Скрытая марковская цепь с тремя состояниями

Зададим требуемую сеть.

In [174]:
```
model = hmm.GaussianHMM(n_components=3, covariance_type="full",
                        algorithm="viterbi")

# начальное рспределение и матрица переходов
model.startprob_ = np.array([1./6, 2./6, 3./6])
model.transmat_ = np.array([
    [1./6, 2./6, 3./6],
    [0.1, 0.4, 0.5],
    [0.1, 0.2, 0.8]
])
# сознательно испортим матрицу переходов, чтобы она была менее тривиальной,
# в сравнении с первым примером

# параметры условных распределений Y_j при условии X_j
model.means_ = np.array([
    [0.0],
    [3.0],
    [6.0]
])
model.covars_ = np.array([
    [[1.]],
    [[1.]],
    [[1.]]
])
```

Сгенерируем некоторую последовательность с помощью определенной выше модели.

In [175]:
```
from matplotlib.colors import ListedColormap
cmap = ListedColormap(["red", "orange", "blue"])  # чтобы было понятно, что к чему

size = 300
Y, X = model.sample(size)  # Y наблюдаемы, X скрытые

plt.figure(figsize=(15, 6))
plt.plot(np.arange(size), Y[:, 0], color='black', alpha=0.1)
plt.scatter(np.arange(size), Y[:, 0], c=np.array(X), cmap=cmap, lw=0, s=60, alpha=0.7)
plt.xlim((0, size))
plt.xlabel("time")
plt.ylabel("Y vars")
plt.show()
```




Обучим скрытую марковскую модель на сгенерированных выше значениях.

In [176]:
```
# создадим модель, для обучения которой будет выполнено <= 100 итераций.
remodel = hmm.GaussianHMM(n_components=3, covariance_type="full",
                          n_iter=100, algorithm="viterbi")

# собственно, обучение
remodel.fit(Y)

# предскажем значения скрытых состояний X для данных значений Y
X_predicted = remodel.predict(Y)
```

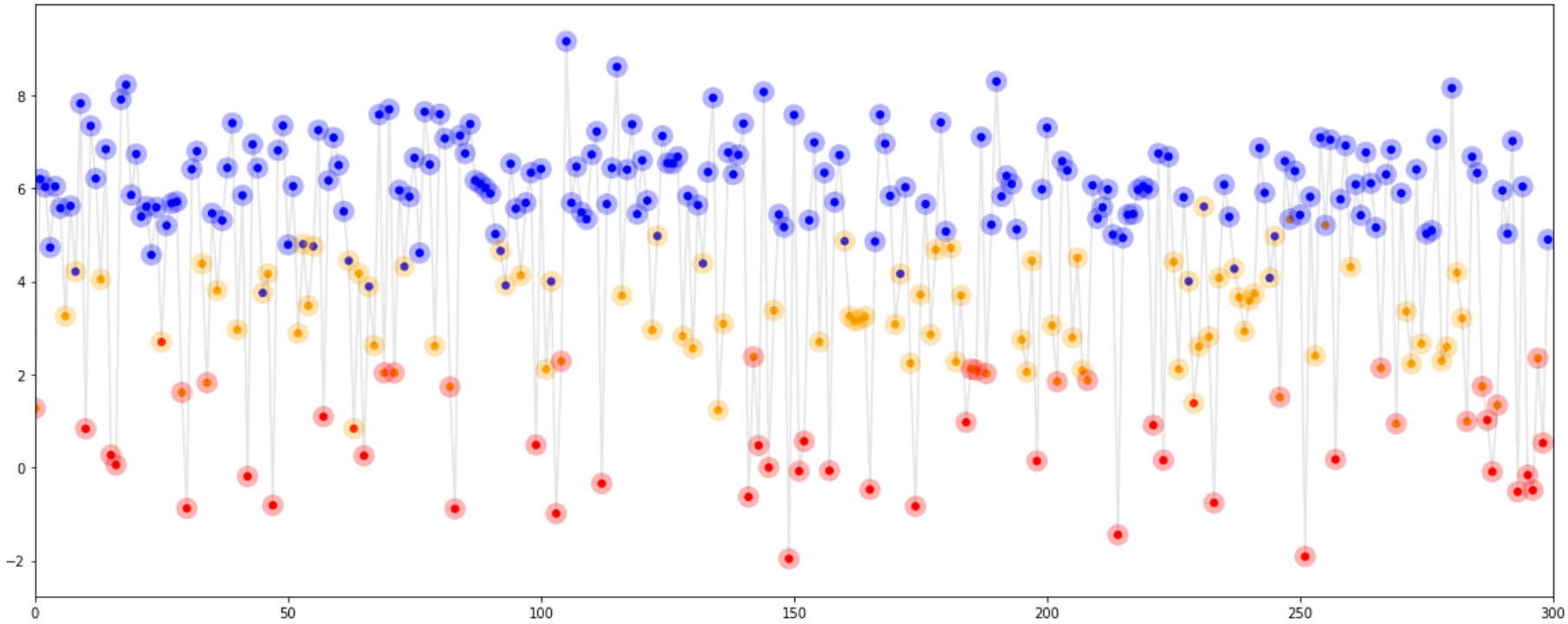
Так как состояния определяются с точностью до перестановки, подберём такую перестановку, чтобы accuracy (доля верных ответов) была максимальна.

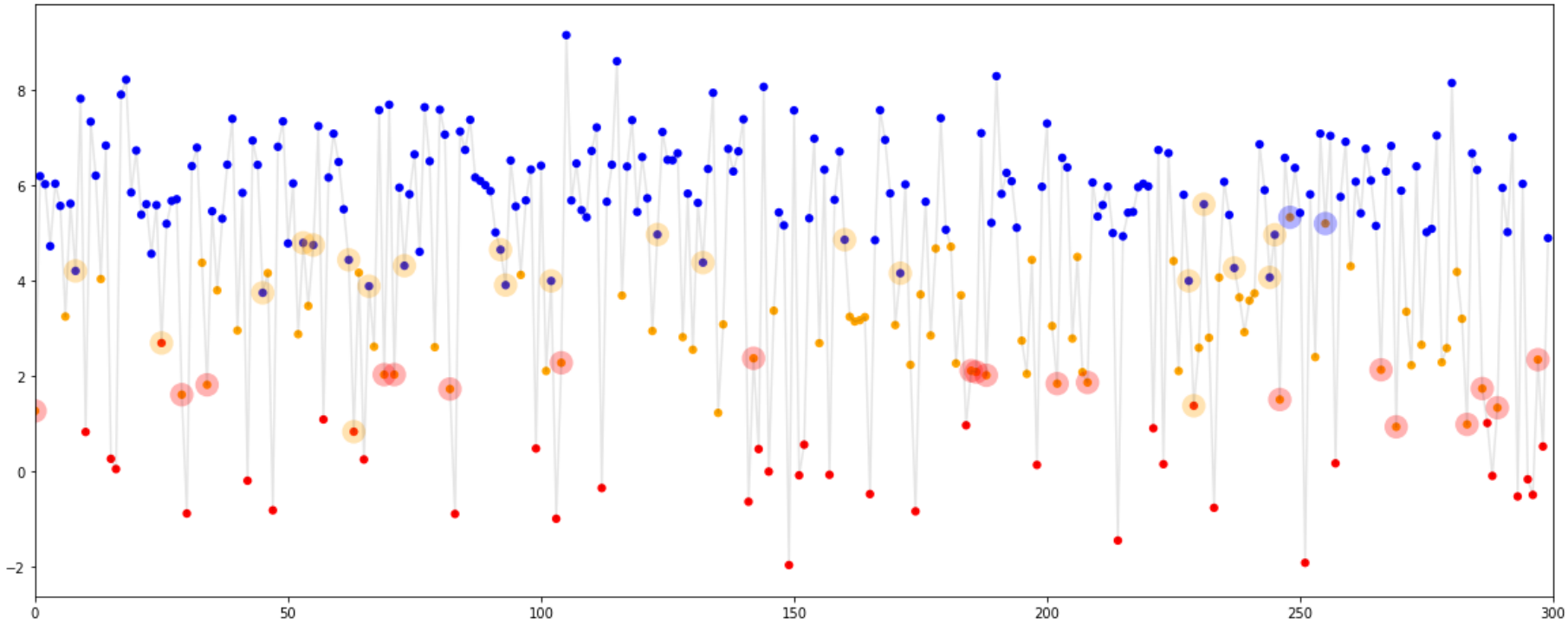
In [177]:
```
import itertools
accuracy_list = []
permutation_list = []
for perm in itertools.permutations([0, 1, 2]):
    accuracy = 0  # считаем не долю, но количество
    for i in range(len(X)):
        accuracy += (perm[X_predicted[i]] == X[i])
    accuracy_list.append(accuracy / len(X))
    permutation_list.append(perm)

print("Accuracy / permutation = ", list(zip(accuracy_list, permutation_list)))
best_permutation = permutation_list[np.array(accuracy_list).argmax()]
print("Best permutation = ", best_permutation)
print("Best accuracy = ", accuracy_list[np.array(accuracy_list).argmax()])
X_predicted_recolored = np.array(list(
    map(lambda x: best_permutation[x], X_predicted)
))

print("X_predicted[:10] = ", X_predicted[:10])
print("X_predicted_recolored[:10]", X_predicted_recolored[:10])
```

```
Accuracy / permutation =  [(0.13, (0, 1, 2)), (0.19333333333333333, (0, 2, 1)), (0.18666666666666668, (1, 0,
 2)), (0.016666666666666666, (1, 2, 0)), (0.85333333333333339, (2, 0, 1)), (0.62, (2, 1, 0))]
Best permutation =  (2, 0, 1)
Best accuracy =  0.853333333333
X_predicted[:10] =  [1 0 0 0 0 0 2 0 2 0]
X_predicted_recolored[:10] [0 2 2 2 2 2 1 2 1 2]
```

Теперь изобразим полученные результаты. На обоих графиках непрозразными маленькими кружочками отмечена исходная последовательность. Полупрозрачными большими кружочками отмечены оценки значений скрытых состояний. На первом графике отмечены все такие точки, на втором только те из них, оценка значения скрытого состояния получилась непрпавильно.

In [178]:
```
colors = np.array(["red", "orange", "blue"])
cmap = ListedColormap(colors)

plt.figure(figsize=(20, 8))
plt.plot(np.arange(size), Y[:, 0], color='black', alpha=0.1)
plt.scatter(np.arange(size), Y[:, 0], c=colors[np.array(X)],
            lw=0, s=40, alpha=1)
plt.scatter(np.arange(size), Y[:, 0], c=colors[np.array(X_predicted_recolored)],
            lw=0, s=250, alpha=0.3)
plt.xlim((0, size))
plt.show()

plt.figure(figsize=(20, 8))
plt.plot(np.arange(size), Y[:, 0], color='black', alpha=0.1)
plt.scatter(np.arange(size), Y[:, 0], c=colors[np.array(X)],
            lw=0, s=40, alpha=1)
plt.scatter(np.arange(size)[X != X_predicted_recolored], Y[:, 0][X != X_predicted_recolored],
            c=colors[np.array(X_predicted_recolored)[X != X_predicted_recolored]],
            lw=0, s=300, alpha=0.3)
plt.xlim((0, size))
plt.show()
```


8
6
4
2
0
−2
0
50
100
150
200
250
300

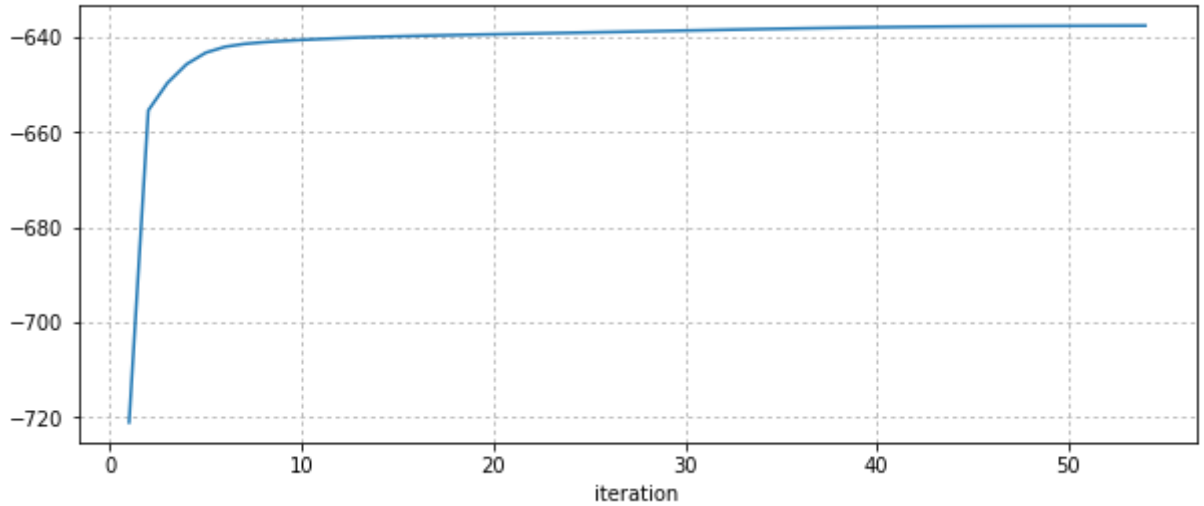



Посмотрим на график (не) сходимости, как в примере.

In [98]:
```
saved_strerr = sys.stderr  # сохраним в переменную поток вывода ошибок
sys.stderr = open('est_values_1.txt', 'w')  # и перенаправим его в файл

# ============
# Для вывода значений функционала нужно поставить параметр verbose
remodel = hmm.GaussianHMM(n_components=3, covariance_type="full",
                          n_iter=100, verbose=True)
remodel.fit(Y)
X_predicted = remodel.predict(Y)
# ============

# Возвращаем все, как было
sys.stderr = saved_strerr
```

Теперь можно загрузить значения и построить график

In [99]:
```
values = np.loadtxt('./est_values_1.txt')

plt.figure(figsize=(10, 4))
plt.plot(values[:, 0], values[:, 1])
plt.xlabel('iteration')
plt.grid(ls=':')
plt.show()
```



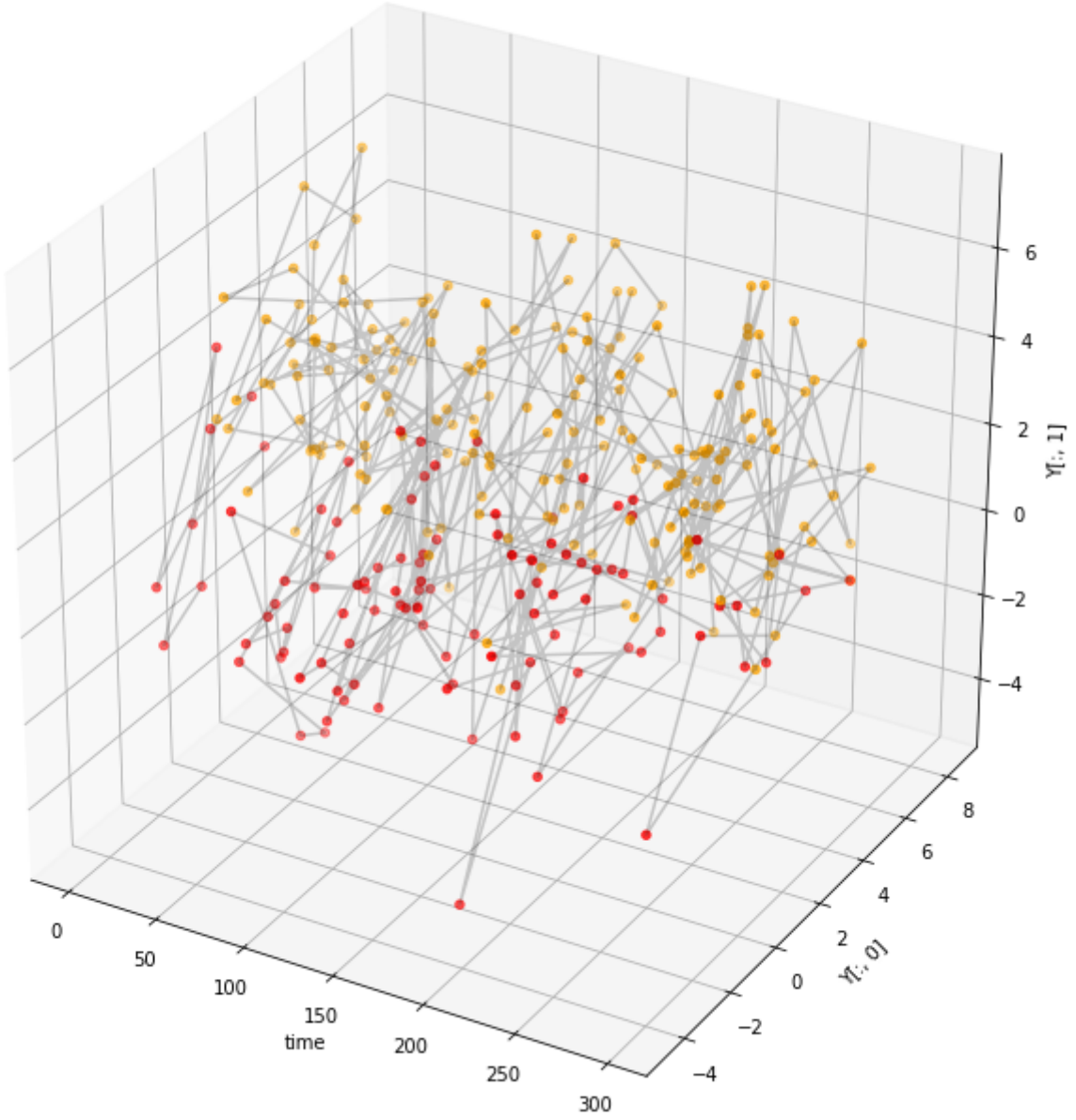

## 1.2 Распределение $Y_j$ при условии $X_j$ является двумерным гассовым

Зададим некоторую скрытую марковскую модель

In [206]:
```
# объявим модель с двумя скрытыми стостояниями
model = hmm.GaussianHMM(n_components=2, covariance_type='full',
                        algorithm='viterbi')

# параметры марковской цепи возьмём из примера
model.startprob_ = np.array([0.6, 0.4])
model.transmat_ = np.array([[0.9, 0.1],
                            [0.07, 0.93]])
# в этом случае матрицу переходов мы портить не будем,
# но дисперсии сделаем побольше, для вывода.

# зададим параметры двумерного распределения
model.means_ = np.array([
    [0.0, 0.0],
    [3.0, 3.0]
])
model.covars_ = np.array([
    [[4., 0.], [0., 4.]],
    [[4., -1], [-1, 4.]]
])
# в этом примере

# неплохо бы проверить положительную определённость матриц.
# воспользуемся критерием Сильвстра
for comp_i in range(model.covars_.shape[0]):
    for i in range(1, model.covars_.shape[1] + 1):
        assert (np.linalg.det(model.covars_[comp_i, :i, :i])) > 0
```

Сгенерируем некоторую последовательность с помощью определенной выше модели и визуализируем её на трёхмерном графике.

In [207]:
```
size = 300
Y, X = model.sample(size)  # Y наблюдаемы, X скрытые

from matplotlib.colors import ListedColormap
cmap = ListedColormap(["red", "orange"])  # чтобы было понятно, что к чему
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(12,12))
ax = fig.gca(projection='3d')
ax.scatter(np.arange(size), Y[:, 0], Y[:, 1], c=np.array(X), cmap=cmap)
ax.plot(np.arange(size), Y[:, 0], Y[:, 1], color='black', alpha=0.2)
ax.set_xlabel("time")
ax.set_ylabel("Y[:, 0]")  # более подробного, но короткого описания не придумал
ax.set_zlabel("Y[:, 1]")
plt.show()
```


0
50
100
150
200
250
300
time
−4
−2
0
2
4
6
8
Y[:, 0]
−4
−2
0
2
4
6
Y[:, 1]

На основе сгенерированной выше последовательности оценим параметры ("обучим") скрытой марковской модели и значения скрытых состояний.

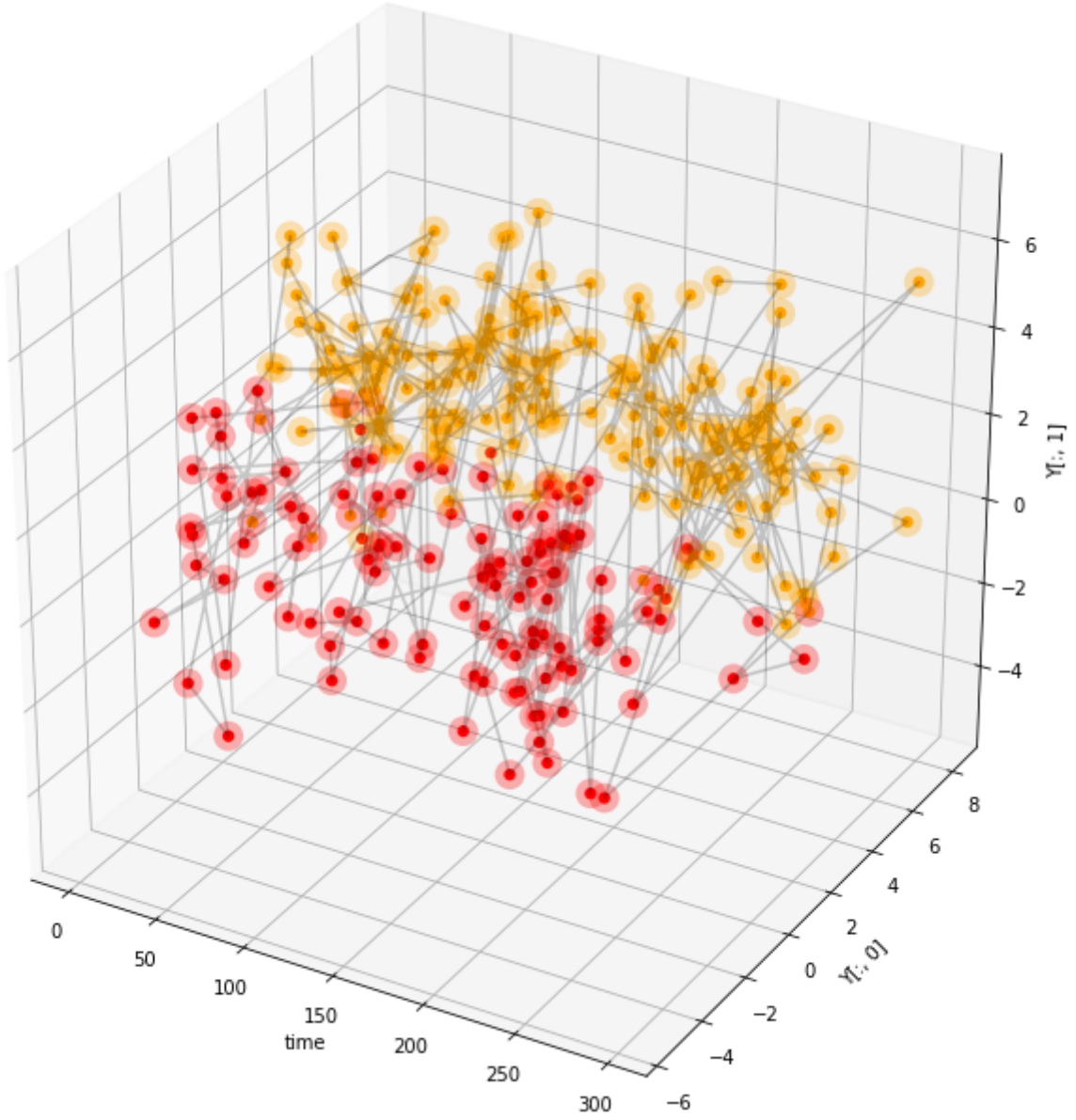
In [208]:
```
# Объявление скрытой марковской модели, в которой при оценке параметров
# будет производиться не более n_iter итераций EM-алгоритма.
remodel = hmm.GaussianHMM(n_components=2, covariance_type="full",
                          n_iter=100, algorithm='viterbi')

# Оценка параметров ("обучение")
remodel.fit(Y)

# Оценка ("предсказание") значений скрытых состояний
X_predicted = remodel.predict(Y)
```

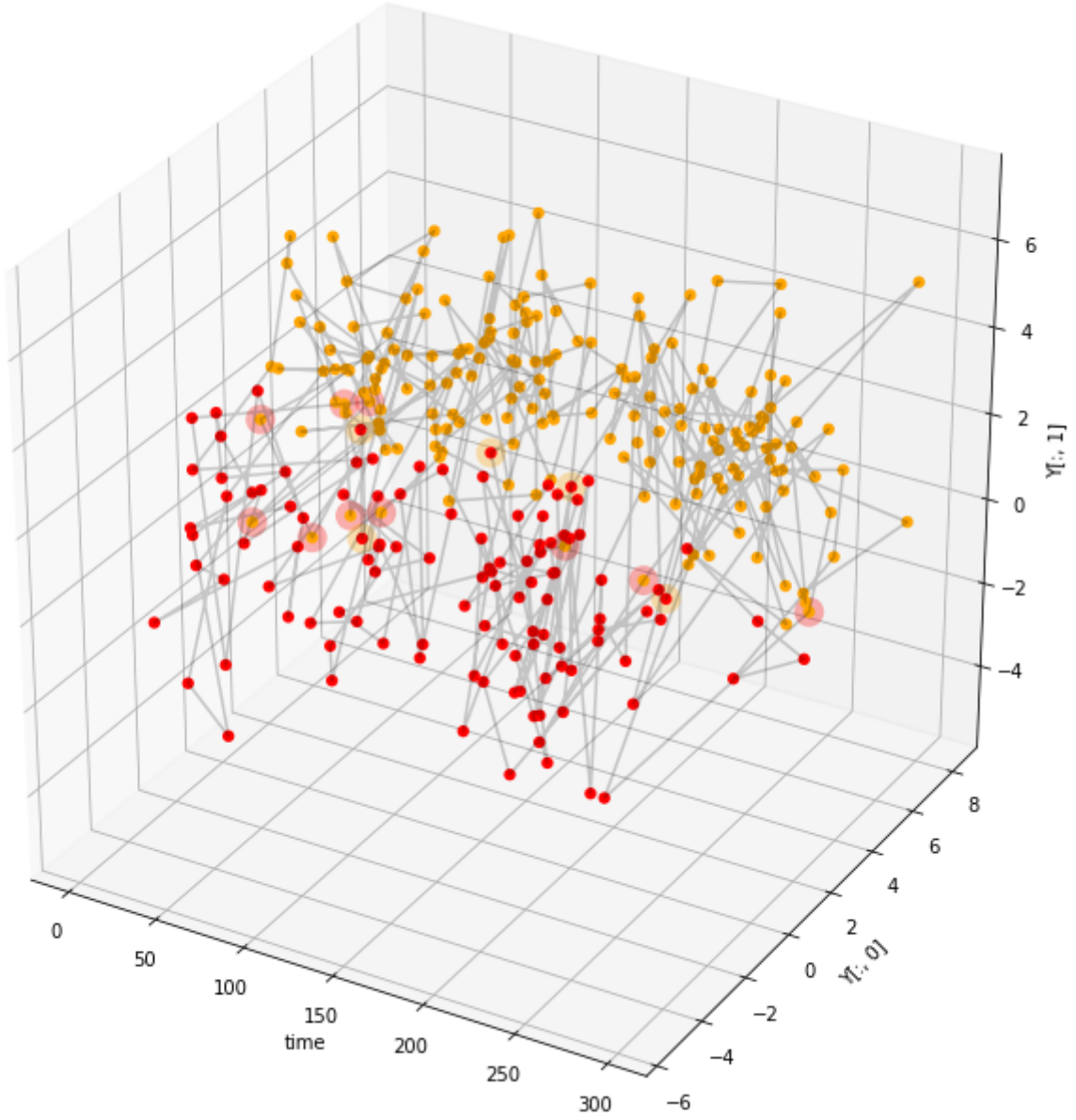
Теперь изобразим полученные результаты. На обоих графиках непрозразными маленькими кружочками отмечена исходная последовательность. Полупрозрачными большими кружочками отмечены оценки значений скрытых состояний. На первом графике отмечены все такие точки, на втором только те из них, оценка значения скрытого состояния получилась непрпавильно.

In [209]:
```
# Состояния определяются с точностью до их перестановки.
# При необходимости меняем местами состояния

from sklearn.metrics import accuracy_score
accuracy = accuracy_score(X, X_predicted)
if accuracy < 0.5:
    X_predicted = 1 - X_predicted
    accuracy = 1. - accuracy
# этот код делает абсолютно то же, что и код в примере


print("Accuracy = ", accuracy)
```

```
Accuracy =  0.976666666667
```

In [187]:
```
colors = np.array(["red", "orange"])
cmap = ListedColormap(colors)

# Первый график
fig = plt.figure(figsize=(12,12))
ax = fig.gca(projection='3d')

ax.plot(np.arange(size), Y[:, 0], Y[:, 1], color='black', alpha=0.2)
ax.scatter(np.arange(size), Y[:, 0], Y[:, 1], c=colors[np.array(X)], cmap=cmap,
           lw=0, s=40, alpha=1)
ax.scatter(np.arange(size), Y[:, 0], Y[:, 1], c=colors[np.array(X_predicted)],
            lw=0, s=250, alpha=0.3)
ax.set_xlabel("time")
ax.set_ylabel("Y[:, 0]")
ax.set_zlabel("Y[:, 1]")
plt.show()

# Второй график
fig = plt.figure(figsize=(12,12))
ax = fig.gca(projection='3d')

ax.plot(np.arange(size), Y[:, 0], Y[:, 1], color='black', alpha=0.2)
ax.scatter(np.arange(size), Y[:, 0], Y[:, 1], c=colors[np.array(X)], cmap=cmap,
           lw=0, s=40, alpha=1)
ax.scatter(np.arange(size)[X != X_predicted],
            Y[:, 0][X != X_predicted],
            Y[:, 1][X != X_predicted],
            c=colors[np.array(X_predicted[X != X_predicted])],
             lw=0, s=250, alpha=0.3)
ax.set_xlabel("time")
ax.set_ylabel("Y[:, 0]")
ax.set_zlabel("Y[:, 1]")
plt.show()
```


0
50
100
150
200
250
300
time
−6
−4
−2
0
2
4
6
8
Y[:, 0]
6
4
2
0
−2
−4
Y[:, 1]

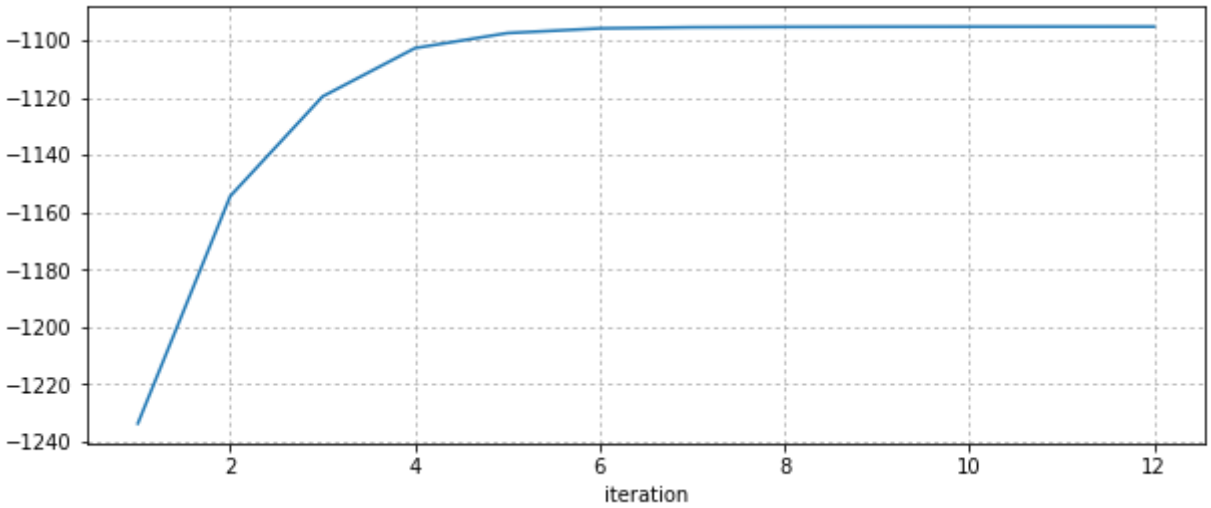

0
50
100
150
200
250
300
time
−6
−4
−2
0
2
4
6
8
Y[:, 0]
6
4
2
0
−2
−4
Y[:, 1]

Посмотрим на график (не) сходимости, как в примере.

In [169]:
```
saved_strerr = sys.stderr  # сохраним в переменную поток вывода ошибок
sys.stderr = open('est_values_2.txt', 'w')  # и перенаправим его в файл

# ============
# Для вывода значений функционала нужно поставить параметр verbose
remodel = hmm.GaussianHMM(n_components=2, covariance_type="full",
                          n_iter=100, verbose=True)
remodel.fit(Y)
X_predicted = remodel.predict(Y)
# ============

# Возвращаем все, как было
sys.stderr = saved_strerr
```

Теперь можно загрузить значения и построить график

In [170]:
```
values = np.loadtxt('./est_values_2.txt')

plt.figure(figsize=(10, 4))
plt.plot(values[:, 0], values[:, 1])
plt.xlabel('iteration')
plt.grid(ls=':')
plt.show()
```




**Дополнительно:** вычислим точность предсказания классов, в предположении, что мы бы знали значения X и пытались использовать логистическую регрессию - интуитивно, по картинке, кажется, легко провести хорошую разделяющую плоскость.

In [216]:
```
from sklearn.linear_model import LogisticRegression
from sklearn import cross_validation
LR = LogisticRegression(random_state=1)
LR.fit(Y, X)
scoring = cross_validation.cross_val_score(LR, Y, X, scoring = 'accuracy', cv = 3)
print("LogisticRegression accuracy = ", scoring.mean())
```

```
LogisticRegression accuracy =  0.856423642364
```

In [ ]:
```
<Код с пояснениями, графики>
```

**Вывод:** Была испытана библиотека hmmlearn.hmm в трёх ситуациях

В первой ситуации марковская цепь $X$ имела простую матрицу переходов - большие ($\geq 0.9$) значения на диагонялях - это означает, что если предыдущее состояние было A, то следующее с высокой вероятностью будет A.

Более того, условные распределения были $\mathcal{N}(0, 1)$ и $n(3, 1)$ - предсказывать состояния $X$, можно было бы и с помощью критерия вида $I[Y > Y.mean() \; 1.5]$, хотя и со значимо меньшей долей верно угаданных состояний, что видно из графика - несколько значений, более близких к 3 (состояние 1), но имеющих предыдущее состояние 0, алгоритм относит к состоянию 0, что он и должен делать, максимизируя правдоподобие. В итоге имеем очень высокую долю угаданных состояний = 0.98. В данных условиях hmm работает почти идеально.

Во второй ситуации была взята менее тривиальная матрица переходов, а пропорции распределений не менялись - $\mathcal{N}(0, 1)$, $n(3, 1)$ и $n(6, 1)$ (шаг - 3, дисперсия - 1). В итоге, состояние 2, для которого вероятность перейти из себя в себя равна 0.8 угадывается достаточно хорошо (а ещё он просто с края, а не по середине, что также упрощает предсказание). При это состояния 0 и 1, для которых зависимости менее тривиальны, определяются хуже. Однако, в том числе, из-за большого числа реальных состояний 2, имеем долю угаданных верно классов = 0.85(3), что (относительно, ибо зависимости действительно менее тревиальны в сравнении с первым примером) является высокой точностью.

В третьей ситуации была взята матрица переходов из первой ситуации, но Y были распредены двумерно, с дисперсией, обеспечивающей перекрытие областей значений Y двух классов, чтобы продемонстрировать ещё раз, что данный алгоритм именно пытается восстановить зависимости цепи X, а не реагирует только на значения, как делал бы метод с индикатором. В итоге - для двумерного случая имеем долю угаданных значений = 0.97(6) [при том, что даже если бы мы знали метки X, и пытались бы использовать логистическую регрессию для предсказания (т.е. выкинули бы информацию о том, что есть какая-то скрытая цепь и пытались бы провести разделяющую гиперплоскость, оптимальную в известном смысле), то получили бы 0.85(6423), что тоже не плохо, но явно значимо уступает почти идеальной hmm]

Кроме гауссовского случая в библиотеке реализовано два других

- `hmm.GMMHMM` --- распределение $Y_j$ при условии $X_j$ является смесью гауссовских распределений,
- `hmm.MultinomialHMM` --- распределение $Y_j$ при условии $X_j$ является дискретным.

## 2. Музыка (предполагалось на семинаре, оцениваться не будет)

Некоторый вспомогательный код.

In [254]:
```
import librosa
from IPython import display

def Audio(url):
    return display.HTML("<center><audio controls><source src='{}'" +
                        "type=\"audio/wav\"></audio>".format(url))
```

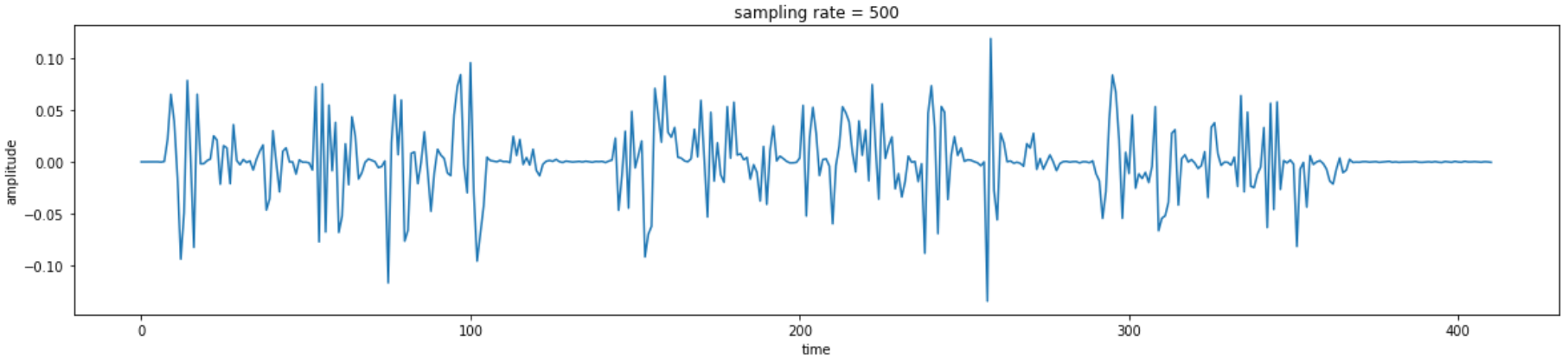
Загрузите сюда некоторый музыкальный трек. Тут же вы его можете послушать.

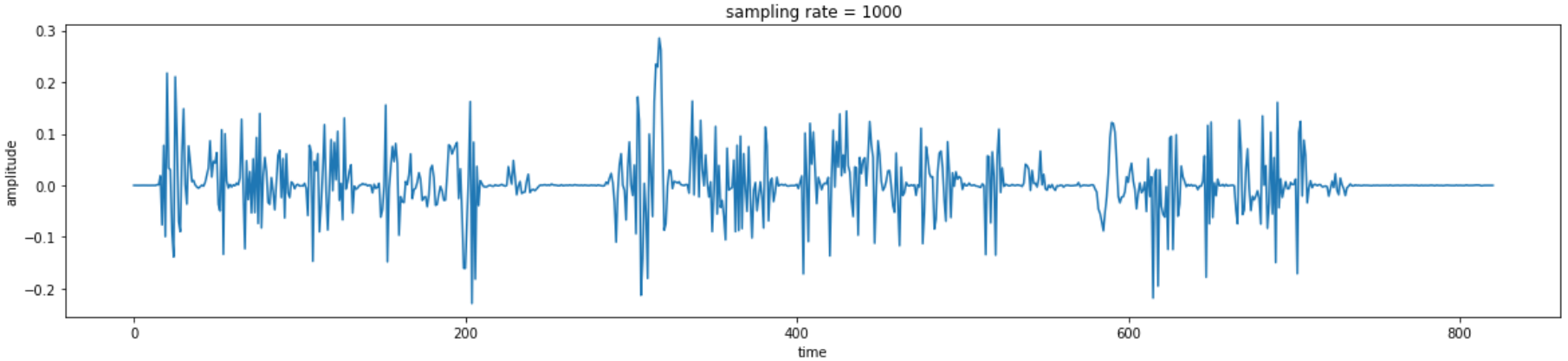
In [255]:
```
sound_file = "track.wav"  # mp3 не пошёл.
Audio(url=sound_file)
```

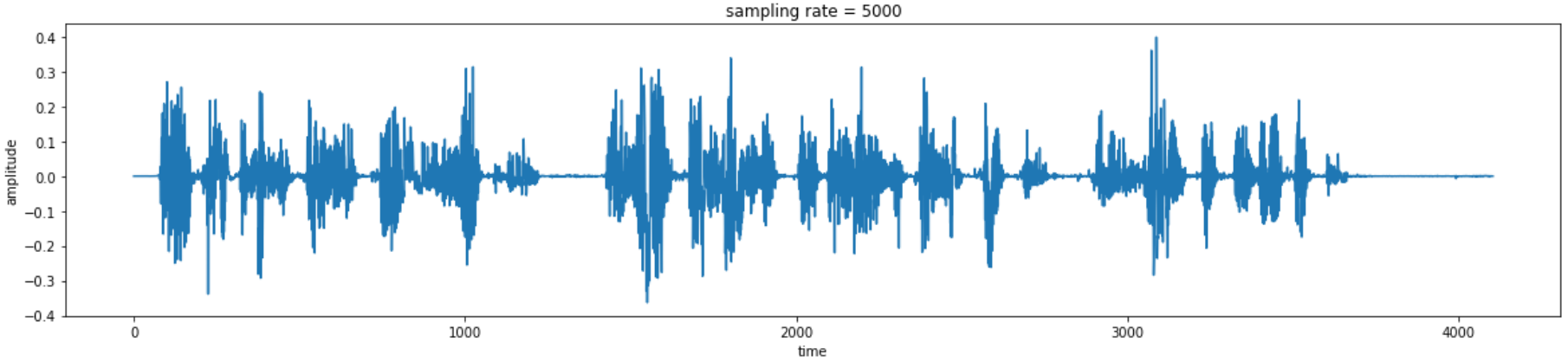
Out[255]:

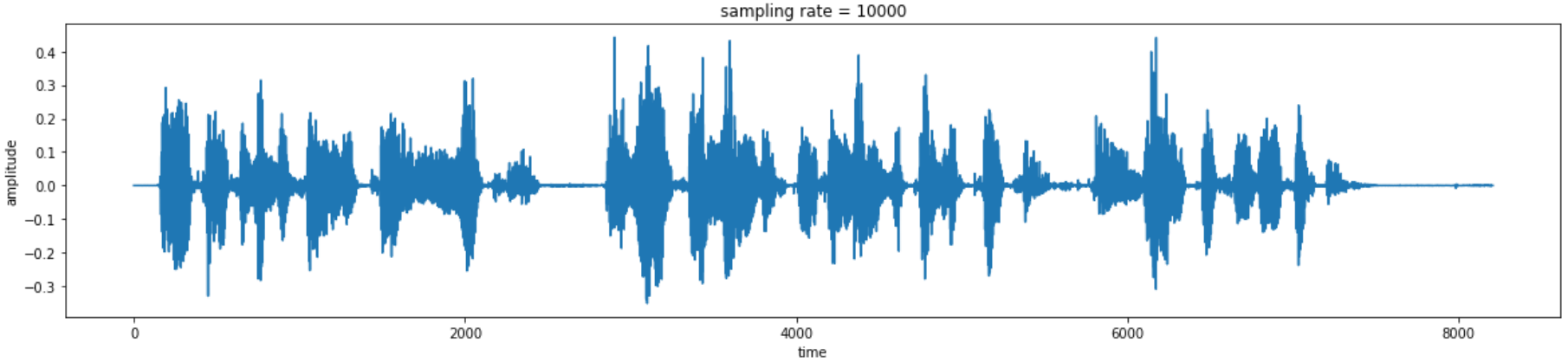

Изобразим графики амплитуды для разной sampling rate (sr).

In [241]:
```
for sr in [500, 1000, 5000, 10000]:
    plt.figure(figsize=(20, 4))
    y, sr = librosa.load(sound_file, sr=sr)
    plt.plot(y[:1000000:10])
    plt.ylabel("amplitude")
    plt.xlabel("time")
    plt.title('sampling rate = {}'.format(sr))
    plt.show()
```






Изобразим спектрограмму

In [245]:
```
sr = 1500

# Если ваш ноутбук не справляется с чем-то, уменьшите параметр n_mels
S = librosa.feature.melspectrogram(y, sr=sr, n_mels=30)
log_S = librosa.logamplitude(S, ref_power=np.max)

plt.figure(figsize=(20,4))
librosa.display.specshow(log_S, sr=sr, x_axis='time',
                         y_axis='mel', cmap='hot')  # ???? Ладно, пропустим.
plt.title('power spectrogram')
plt.colorbar(format='%+02.0f dB')
plt.tight_layout()

# не пошло, ну и ладно :(
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-245-8734151ba3af> in <module>()
      6
      7 plt.figure(figsize=(20,4))
----> 8 librosa.display.specshow(log_S, sr=sr, x_axis='time',
      9                          y_axis='mel', cmap='hot')  # ???? Ладно, пропустим.
     10 plt.title('power spectrogram')

AttributeError: module 'librosa' has no attribute 'display'


<matplotlib.figure.Figure at 0x7f18470364e0>
```

Если выполнить операцию транспонирования, то мы получим некоторый многомерный случайный процесс. К нему уже можно применить скрытую марковскую модель.

In [ ]:
```
Y = log_S.T
```

Соберите коллекцию музыкальных треков одного стиля или же одного исполнителя.

**Варианты задания:**

1. Объедините спектрограммы всех треков коллекции в одну большую спектрограмму и выполните на ней обучение скрытой марковской модели. Сколько итераций потребовалось EM-алгоритму для сходимости? Какие по смыслу состояния были обнаружены? Примените модель к новым трекам того же стиля.
2. Вручную разбейте все треки на припев и все остальное, это будут скрытые состояния. Оцените вручную все параметры модели. Примените модель к новым трекам. Какое получается качество?

## 3. Part-of-speach tagging (8 баллов)

Теперь вам нужно самостоятельно реализовать метод распознавания частей речи для слов в предложении на основе скрытых марковских моделей. Метод реализовать можно с помощью библиотеки `hmmlearn`, но хранение разреженных матриц в в формате обычных матриц потребует большого количества памяти, поэтому на этот раз реализовать HMM придется самостоятельно.

Будем считать, что каждый следующий тег непосредственно зависит только от двух предыдущих. Такая модель является марковской цепью, если в качестве состояний рассматривать все возможные пары тегов, причем матрица переходных вероятностей будет разреженной. Чтобы однозначно задать матрицу переходных вероятностей достаточно определить вероятности $\mathrm{P}(X_3 = tag_3 | X_2 = tag_2, X_1 = tag_1)$, в качестве которых возьмем их оценку максимального правдоподобия (см. семинар). Сделайте так же оценку вероятностей $\mathrm{P}(Y_i = word | X_i = tag)$ для определения распределения слов для каждого тега.

Для удобства реализации можно считать, что перед началом предложения и после конца предложения находится несколько "пустых" тегов.

После того, как определена модель, нужно реализовать функцию, которая по заданному предложению для данной модели будет находить траекторию Витерби. Эта траектория и будет являться оценкой последовательности тегов для данного предложения. Посчитайте точность определения тегов на тестовом наборе данных.

**Данные** возьмите такие же, как в примере с семинара.

Сравните точность вашей модели с простой моделью, которая была показана на семинаре, а так же со следующими встроенными моделям:

---

**в этом задании сначала идут отрывки кода модели, снабжённые комментариями, в конце - модель, собранная в самостоятельный класс и оптимизированная по времени (test_sents за 6 минут, accuracy 94.8%, ~~самая быстрая модель на диком западе~~ ), следовательно, трудночитаемая.**

Загрузим данные

In [115]:
```
import numpy as np
import nltk  # sudo pip3 install nltk
from collections import Counter, defaultdict
from sklearn.metrics import accuracy_score
```

In [116]:
```
# nltk.download()
```

In [117]:
```
from nltk.corpus import conll2000
```

По-хорошему, надо использовать train_test_split или хотя бы перемешать данные, ибо такое тестирование потенциально некорректно (а вдруг предложения как-то отсортированы), но раз данные как в примере с семинара - значит данные как в примере с семинара.

In [118]:
```
train_sents = conll2000.tagged_sents()[:8000]
test_sents = conll2000.tagged_sents()[8000:]
```

In [119]:
```
train_sents
```

Out[119]:
```
[[('Confidence', 'NN'), ('in', 'IN'), ('the', 'DT'), ('pound', 'NN'), ('is', 'VBZ'), ('widely', 'RB'), ('exp
ected', 'VBN'), ('to', 'TO'), ('take', 'VB'), ('another', 'DT'), ('sharp', 'JJ'), ('dive', 'NN'), ('if', 'I
N'), ('trade', 'NN'), ('figures', 'NNS'), ('for', 'IN'), ('September', 'NNP'), (',', ','), ('due', 'JJ'),
 ('for', 'IN'), ('release', 'NN'), ('tomorrow', 'NN'), (',', ','), ('fail', 'VB'), ('to', 'TO'), ('show', 'V
B'), ('a', 'DT'), ('substantial', 'JJ'), ('improvement', 'NN'), ('from', 'IN'), ('July', 'NNP'), ('and', 'C
C'), ('August', 'NNP'), ("'s", 'POS'), ('near-record', 'JJ'), ('deficits', 'NNS'), ('.', '.')], [('Chancello
r', 'NNP'), ('of', 'IN'), ('the', 'DT'), ('Exchequer', 'NNP'), ('Nigel', 'NNP'), ('Lawson', 'NNP'), ("'s",
 'POS'), ('restated', 'VBN'), ('commitment', 'NN'), ('to', 'TO'), ('a', 'DT'), ('firm', 'NN'), ('monetary',
 'JJ'), ('policy', 'NN'), ('has', 'VBZ'), ('helped', 'VBN'), ('to', 'TO'), ('prevent', 'VB'), ('a', 'DT'),
 ('freefall', 'NN'), ('in', 'IN'), ('sterling', 'NN'), ('over', 'IN'), ('the', 'DT'), ('past', 'JJ'), ('wee
k', 'NN'), ('.', '.')], ...]
```

К сожалению, не все предложения заканчиваются точкой, что немного портит нам жизнь.

In [120]:
```
for sent in conll2000.tagged_sents():
    if (sent[-1][0] != "."):
        print(sent)
        break
```

```
[('Thursday', 'NNP'), (',', ','), ('he', 'PRP'), ('reminded', 'VBD'), ('his', 'PRP$'), ('audience', 'NN'),
 ('that', 'IN'), ('the', 'DT'), ('government', 'NN'), ('``', '``'), ('can', 'MD'), ('not', 'RB'), ('allow',
 'VB'), ('the', 'DT'), ('necessary', 'JJ'), ('rigor', 'NN'), ('of', 'IN'), ('monetary', 'JJ'), ('policy', 'N
N'), ('to', 'TO'), ('be', 'VB'), ('undermined', 'VBN'), ('by', 'IN'), ('exchange', 'NN'), ('rate', 'NN'),
 ('weakness', 'NN'), ('.', '.'), ("''", "''")]
```

Посмотрим на число различных тегов в обучающей выборке (чтобы понять, что ставить неизвестным словам, например). Кстати, для тега, разделяющего предложения подходит точка. (см данные, да, точка - тег)

In [121]:
```
tags_summary = dict()
for sent in train_sents:
    for word, tag in sent:
        if tag in tags_summary:
            tags_summary[tag] += 1
        else:
            tags_summary[tag] = 1
print(tags_summary)
```

```
{'TO': 4523, 'PDT': 48, '#': 32, '(': 235, 'WP$': 33, 'VBG': 2956, 'VB': 5443, 'RBS': 172, '.': 7899, 'POS':
 1575, 'SYM': 6, ',': 9639, 'JJ': 11779, 'NNPS': 372, 'WRB': 433, 'UH': 15, 'WDT': 871, 'MD': 1950, ')': 24
2, 'PRP$': 1700, 'VBZ': 4291, 'NN': 27005, 'NNS': 12282, 'VBN': 4238, 'PRP': 3510, 'WP': 488, 'FW': 37, 'C
D': 7070, 'JJR': 767, 'VBD': 5892, 'EX': 185, 'DT': 16556, ':': 932, 'RP': 70, 'RB': 6010, 'CC': 4855, 'RB
R': 301, 'JJS': 338, '$': 1534, 'IN': 20432, '``': 1396, "''": 1360, 'NNP': 17579, 'VBP': 2651}
```

Чаще всего встречаются NN, собственно, в примере, неизвестным словам давался NN.

Сделаем функцию, которая будет приводить слова к "нормальному" виду. В частности, много слов, которые присутствуют в тестовой выборке, но отсутствуют в обучающей - это числа и имена собственные.

С помощью регулярного выражения мы будем проверять, является ли слово числом, и если да - заменять все такие на единственное слово-число, т.к. смысла считать разные числа разными словами для определения чати речи нет. Дополнительно можно было бы что-то сделать с именами собственными, т.к. их можно выделять по заглавным буквам, но здесь это не реализовано, т.к. по accurary,

подсчитаному на первых 50 предложениях из тестовой выборки, такое преобразование слабо влияет на результат в текущей модели. [по-хорошему, для таких проверок нужно выделить три подвыборки - первая для нахождения параметров СММ, вторая - для выбора гиперпараметров, таких как эти преобразования, и третья - для поиска итогового accuracy, чтобы избегать переобучения.]

In [122]:
```
import re
number_re = re.compile(r"\d+\.?\d*")  # регулярное выражение для чисел
def word_prepare(word, first_in_sentense):
    upper_case = word.istitle()  # проверка, что первая буква - завглавная
    word = word.lower()
    re_res = number_re.match(word)
    if (not re_res is None) and (re_res.span() == (0, len(word))):
        # print(word)
        return "17"  # заменяем все числа на фиксированное число.
                     # это увеличивает accuracy на test_sent[:50]
                     # на 1.2% и уменьшает словарь на ~ 1300 слов

    if (not first_in_sentense) and (upper_case):
        return "Julia"  # если слово в середине предложения написано
                        # с большой буквы - это имя собственное
                        # заменим все обнаруженные на какое-то одно

    return word

word_prepare("1.72", False), word_prepare("vector", False)
```

Out[122]:
```
('17', 'vector')
```

Составим словарь всех слов, чтобы работать с цифрами, а не строками. Т.е. каждому слову из обучающей выборки сопоставим число.

In [123]:
```
words = dict()
for sent in train_sents:
    for i, (word, tag) in enumerate(sent):
        word = word_prepare(word, i==0)
        # конечно, confidence и Confidence - одно слово
        total = len(words)
        if not (word in words):
            words[word] = total

len(words)
# без замены всех чисел на одно было ~16300 слов, стало ~15000
# дополнительно, после замены всех имён собственных на одно, стало 12665
```

Out[123]:
```
12665
```

Аналогично - для тегов.

In [124]:
```
tags = dict()
for sent in train_sents:
    for word, tag in sent:
        total = len(tags)
        if not (tag in tags):
            tags[tag] = total

print(tags, len(tags))
```

```
{'TO': 6, 'PDT': 37, '#': 24, '(': 25, 'WP$': 41, 'VBG': 16, 'VB': 7, 'RBS': 36, '.': 14, 'POS': 13, 'SYM':
 42, ',': 11, 'JJ': 8, 'NNPS': 28, 'WRB': 35, 'UH': 43, 'WDT': 34, 'MD': 23, ')': 27, 'PRP$': 17, 'VBZ': 3,
 'NN': 0, 'NNS': 9, 'VBN': 5, 'PRP': 29, 'WP': 31, 'FW': 40, 'CD': 18, 'JJR': 33, 'VBD': 21, 'EX': 22, 'DT':
 2, ':': 39, 'RP': 38, 'RB': 4, 'CC': 12, 'RBR': 32, 'JJS': 30, '$': 26, 'IN': 1, '``': 19, "''": 20, 'NNP':
 10, 'VBP': 15} 44
```

Сделаем две маленькие, но приятные функции, кодирующие - декодирующие пару тегов

In [125]:
```
def encode_tag_pair(t1, t2):
    return t1 * len(tags) + t2

def decode_tag_pair(code):
    return int(code) // len(tags), int(code) % len(tags)
```

Оформим выборку в массивы - y будет содержать слова предложений, X - состояния марковской модели как пары тегов в явном виде. При этом будем считать, что перед первым словом в предложении шла точка.

In [139]:
```
sep = "."  # разделитель предложений
sep_id = tags[sep]

y = []
for sent in train_sents:
    for i, (word, tag) in enumerate(sent):
        y.append(words[word_prepare(word, i==0)])
    # y.append(sep_id)


X = []
last_tag = sep_id  # точка - это тег
for sent in train_sents:
    for word, tag in sent:
        tag_id = tags[tag]
        X.append(encode_tag_pair(last_tag, tag_id))
        last_tag = tag_id
    # X.append(encode_tag_pair(last_tag, sep_id))
    # last_tag = sep_id

print(y[:5], len(y))  # логично, некоторый префикс есть 0, 1, 2 ...
print(X[:5], len(X))  # на втором месте стоит, очевидно, 0 * len(T) + 1
```

```
[0, 1, 2, 3, 4] 189702
[616, 1, 46, 88, 3] 189702
```

Вычислим некоторые константы.

In [140]:
```
r = len(tags) * len(tags)  # число состояний скрытой цепи Маркова ~ 2e3
T = len(X)  # длина выборки, на которой ищем ОМП ~ 2e5
Y_max = len(words)  # число принимаемых Y значений ~ 2E4

r, T, Y_max
```

Out[140]: (1936, 189702, 12665)

Оценим условные вероятности $P(Y_i = w|X_i = t)$ частотным методом. Т.е.

$$P(Y_i = w|X_i = t) = \frac{P(Y_i = w, X_i = t)}{P(X_i = t)} := \frac{Count(Y_i = w, X_i = t)}{Count(X_i = t)} = \frac{Count(Y_i = w, X_i = t)}{\sum_w Count(Y_i = w, X_i = t)}$$

In [141]:
```
condition_prob_matrix = np.zeros((Y_max + 1, len(tags)))  # матрица переходов
# все пары ненулевых элементов prob_matrix, каждая по одному разу

for i in range(len(X)):
    t1, t2 = decode_tag_pair(X[i])
    condition_prob_matrix[y[i], t2] += 1  # считаем сумму I(Yi =w, Xi = t)

condition_prob_matrix[Y_max, :] = condition_prob_matrix[:Y_max, :].mean(axis=0)
# для неизвестных слов. В качестве альтернативы можно было заменять неизвестное
# слово на известное, например, this - 0.932716568545
# проставить равномерные условные вер-ти -  0.942809083263
# сделать, как выше - 0.956265769554
# (оценки accuracy по test_sents[:50] - выше говорилось,
# почему для этого стоило отвести отдельную подвыборку)

cnt_xi = condition_prob_matrix.sum(axis = 0)  # сумма I(Xi = t)

for y_value in range(Y_max + 1):
    for tag in range(len(tags)):
        condition_prob_matrix[y_value, tag] /= cnt_xi[tag]

assert ((np.abs(condition_prob_matrix.sum(axis=0) -
                np.ones(len(tags))) < 1e-9).sum() == len(tags))
# проверяем, что получились одни единицы
```

Оценим переходные вероятности методом наибольшего правдоподобия, как говорилось на семинаре.

In [142]:
```
def fit_prob_matrix(coeff=1., log=False):
    nij = np.zeros((r, r), dtype=float)  # nij = sum_k I(X_{k-1} =i, X_k = j)
    ni = np.zeros(r, dtype=int) #ni = sum_j nij

    for i in range(len(X) - 1):  # собственно, считаем вхождения пар
        nij[X[i], X[i + 1]] += 1

    ##################
    ### Для того, чтобы все тройки были возможны, добавим константу к
    ### разрешённым переходам в матрице, иначе - некоторые тройки станут
    ### недопустимыми и на новых предложениях будет плохой прогноз.
    ### !!! Ниже будет обоснован выбор константы !!!
    for t1 in range(len(tags)):
        for t2 in range(len(tags)):
            for t3 in range(len(tags)):
                x1 = encode_tag_pair(t1, t2)
                x2 = encode_tag_pair(t2, t3)
                nij[x1][x2] += coeff
    ##################

    ni = nij.sum(axis = 1)

    global prob_matrix
    prob_matrix = np.zeros((len(tags), len(tags), len(tags)))
    # prob_matrix[t1, t2, t3] = P(X_{i+1} = (t2, t3)| X_i = (t1, t2))

    for t1 in range(len(tags)):
        for t2 in range(len(tags)):
            i = encode_tag_pair(t1, t2)
            for t3 in range(len(tags)):
                j = encode_tag_pair(t2, t3)
                prob_matrix[t1, t2, t3] = (
                    nij[i, j] / ni[i] if nij[i, j] > 0 else 0
                )

    if log:
        print(prob_matrix.sum(axis = 2))
        # должны быть 1 для достижимых состояний и 0 для недостижимых
        # (т.е. таких пар, которых не встретилось в обучающей выборке)

    # проверим, что корректно подсчиталось
```

```
    for line in prob_matrix.sum(axis = 2):
        for val in line:
            assert (abs(val -1.) < 1e-9) or (abs(val - 0.) < 1e-9)

    valid_states = np.arange(r)[(ni > 0)]
    # вычислим достижимые состояния, чтобы рассматривать далее только их
    if log:
        print(valid_states, len(valid_states))

fit_prob_matrix()
```

В конце-концов оценим начальное распределение. Для этого посмотрим в данные и вычислим частотным образом вероятности появления какого-то тега в начале предложения. Предложения разделяются точкой, так что все начальные состояния будут иметь вид encode_tag_pair(tags["."], ?)

In [143]:
```
start_probs =  np.zeros(len(tags))
print("Training sentenses count = ", len(train_sents))
for sent in train_sents:
    start_probs[tags[sent[0][1]]] += 1

start_probs += 1. / len(train_sents)

print(start_probs.sum())
start_probs /= start_probs.sum()
print(start_probs, start_probs.sum())
```

```
Training sentenses count =  8000
8000.0055
[  4.54999843e-02   1.24874930e-01   2.12624869e-01   1.75001442e-03
   6.14999733e-02   4.87501227e-03   3.62501313e-03   5.12501210e-03
   4.39999854e-02   4.27499862e-02   1.90624885e-01   1.56249893e-08
   5.68749765e-02   1.56249893e-08   1.56249893e-08   3.75015367e-04
   1.38750061e-02   8.87500952e-03   1.63750044e-02   7.33749652e-02
   3.75015367e-04   1.25001477e-03   3.25001339e-03   6.25015195e-04
   1.56249893e-08   3.37501330e-03   1.56249893e-08   1.25015539e-04
   2.50015453e-04   6.03749741e-02   5.00001219e-03   3.62501313e-03
   2.37501399e-03   1.00001494e-03   6.25015195e-04   7.00001081e-03
   1.56249893e-08   1.56249893e-08   1.56249893e-08   2.37501399e-03
   1.56249893e-08   1.25015539e-04   5.00015281e-04   7.50015109e-04] 1.0
```

Итого, мы оценили переходные и условные вероятности. Теперь мы можем искать траекторию Витерби. Реализованный ниже алгоритм по названиям переменных пытается быть похожим на алгоритм, описанный на семинаре. Однако, можно было реализовывать его без взятия логарифма, заменяя сложение умножением, как описано на викиконспектах. По-идее такой вариант должен работать быстрее.

Здесь реалозован именно тот вариант, что был описан на семинаре, с сохранением обозначений, где можно.

Так как переходы из состояния (a, b) в (c, d), где b != c запрещены (вер-ти перехода равны нулю), то и матрицы мы будем хранить не полностью, а только для допустимых пар состояний. Например, матрица перехода будет иметь три индекса prob_matrix[t1, t2, t3], что эквивалентно prob_matrix[(t1, t2), (t2, t3)], если бы мы хранили матрицу полностью. Так мы уменьшим затраты памяти в len(tags) = 44 раз.

Условная вероятность при условии пары (t1, t2) равна вероятности при условии t2, т.е.
$P(Y_i = y_i | X_i = x_i = (t1, t2)) = P(Y_i = y_i | X_i[1] = x_i[1] = t2)$. Иначе говоря, при фиксированной части речи слова i вероятность слова быть y_i не зависит от части речи слова (i-1). (повторяется предложенная в условии идея)

In [144]:
```
from tqdm import tqdm_notebook, tnrange  # не заработал.(
from tqdm import tqdm
_unknown_word_id = Y_max

def my_log(f):
    if f != 0:
        return np.log(f)
    else:
        return 0. - np.inf


def eval_viterbi(y, log=False):
    T = len(y)

    g1 = np.zeros((len(tags), len(tags)))
    # g1 из презентации для вершины x = (t1, t2)

    for t1 in range(len(tags)):
        for t2 in range(len(tags)):
            g1[t1, t2] = (my_log(start_probs[t2]) +
                          my_log(condition_prob_matrix[y[0], t2]))


    g = np.zeros((T, len(tags), len(tags), len(tags)))
    # g из презентации, где g[T, t1, t2, t3] соответствует переходу из
    # состояния x1 = (t1, t2) в состояние x2 = (t2, t3) во время T (y[T])
    # это нужно, т.к. переходы из (a, b) в (c,d), где b != c невозможны,
    # а хранение лишней памяти - растратно и медленно (промахи кэша и выделение памяти)

    """
    for t1 in range(len(tags)):
        for t2 in range(len(tags)):
            for t3 in range(len(tags)):
                g[0, t1, t2, t3] = g1[t2, t3]
    """
    # Код снизу делает то же самое, что и код сверху
    for t1 in range(len(tags)):
        g[0, t1, :, :] = g1

    log_prob_matrix = np.log(prob_matrix)
    # один вызов для всей матрицы раз в пять ускоряет работу,
```

```
# в сравнении с множеством вызовов для одного числа

for t in range(T):
    if (log):
        print(t, end=",")
    for t3 in range(len(tags)):
        v2 = my_log(condition_prob_matrix[y[t], t3])
        for t2 in range(len(tags)):
            for t1 in range(len(tags)):
                v1 = log_prob_matrix[t1, t2, t3]
                g[t, t1, t2, t3] = v1 + v2


################
if (log):
    print("G_dp")

G_dp_vals = np.zeros((T, len(tags), len(tags)))   # G* в презентации
# состояния динамики кодируем так же - G_dp_vals[T, t1, t2]
# соответствует G_dp[T, x], где x = (t1, t2)
# - уменьшим расход памяти в len(tags) раз.

G_dp_ways = np.zeros((T, len(tags), len(tags)), dtype=int)
# будем запонимать, откуда пришли, чтобы восстанавливать ответ проще.

for t2 in range(len(tags))[::-1]:  # инициализация ДП
    for t3 in range(len(tags)):
        G_dp_vals[0, t2, t3] = g1[t2, t3]
        G_dp_ways[0, t2, t3] = -1
        # храним пару - значение и номер ячейки, откуда пришли

for t in range(1, T):
    if (log):
        print(t, end=",")

    for t2 in range(len(tags)):
        for t3 in range(len(tags)):
            possible_values = (G_dp_vals[t - 1, :, t2] + g[t, :, t2, t3])
            # на месте двоеточия - индекс, который мы
            # перебираем, максимизируя результат
            argmax = (G_dp_vals[t - 1, :, t2] + g[t, :, t2, t3]).argmax()
            G_dp_vals[t, t2, t3] = possible_values[argmax]
```

```
                G_dp_ways[t, t2, t3] = argmax
                # вычислим максимум с помощью питру, это быстрее

    #################


    argmax = G_dp_vals[T - 1, :, :].argmax()
    k_star = (argmax // len(tags), argmax % len(tags))

    if (log):
        print((G_dp_vals[T-1, :, :] != -np.inf).sum())

    ans = [-1] * T
    for t in range(T)[::-1]:
        ans[t] = k_star[1]
        previous_tag = G_dp_ways[t, k_star[0], k_star[1]]
        k_star = (previous_tag, k_star[0])

    return ans

def score_me(test_sent, log=False):
    if log:
        print("Score_me = ", test_sent)
    sent = [word for word, tag in test_sent]
    sent_tag = [tag for word, tag in test_sent]

    y = []
    for i, word in enumerate(sent):
        word_prep = word_prepare(word, i==0)
        if word_prep in words:
            y.append(words[word_prep])
        else:
            if log:
                print("fuck:", word)
            y.append(_unknown_word_id)
    if log:
        print("y = ", y)

    X = []
    for tag in sent_tag:
        X.append(tags[tag])
    if log:
```

```
        print("X = ", X)

    X_predict = eval_viterbi(y)
    if log:
        print("X_predict = ", X_predict)
    acc_score = accuracy_score(X_predict, X)
    if log:
        print("Accuracy = ", acc_score)
    return ((np.array(X_predict) == np.array(X)).sum(), len(X))
```

Оценим работоспособность алгоритма на первых 50 предложениях из тестовой выборки

In [145]:
```
print(len(test_sents))
```

```
2948
```

In [146]:
```
%%time
accs = np.array([score_me(test_sents[i])
                 for i in range(50)  # предложенный отладочный вывод не заработал
                 if (print(i, end=",") or True)])
print("Accuracy on sents[i] = ", accs[:, 0] / accs[:, 1])
total_accuracy = np.array(accs[:, 0].sum() / accs[:, 1].sum()).mean()
print("Total accuracy = ", total_accuracy)
```

```
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,3
9,40,41,42,43,44,45,46,47,48,49,Accuracy on sents[i] =  [ 0.94117647  1.          1.          0.92857143  1.
          0.81818182
  0.96153846  1.          0.96666667  1.          0.93939394  1.          1.
  1.          1.          1.          1.          1.          0.97058824
  0.90909091  0.90909091  1.          0.9375      0.96296296  0.94117647
  1.          1.          1.          0.92592593  0.90909091  1.          1.
  0.89285714  1.          0.95238095  0.90909091  1.          0.9375
  0.86666667  0.84375     0.96        0.95238095  1.          0.96428571
  0.9047619   1.          0.95238095  0.9047619   0.9047619   1.        ]
Total accuracy =  0.957106812447
CPU times: user 1min 26s, sys: 264 ms, total: 1min 26s
Wall time: 1min 26s
```

Получили хорошее значение. Теперь реализуем код в класс, который будет работать ~ в 6 раз быстрее и на нём обоснуем выбор константы.

---

# Готовый класс

### Внимание ! хоть код в классе и эквивалентен коду выше, он был оптимизирован (раз в 6 по скорости на test_sents[:50]) и стал менее читаем

In [147]:
```
import numpy as np
import nltk  # sudo pip3 install nltk
from collections import Counter, defaultdict
from sklearn.metrics import accuracy_score
from nltk.corpus import conll2000
import re

train_sents = conll2000.tagged_sents()[:8000]
test_sents = conll2000.tagged_sents()[8000:]
```

In [148]:
```
class HMMTagger:
    """Алгоритм, позволяющий вычислять теги для слов,
    учитывая зависимость от тега предыдущего слова.
    Ищет траекторию Витерби для скрытой Марковской
    цепи, чью параметры оценены по ОМП.
    """

    def __init__(self, train_sents, coeff=1):
        HMMTagger.number_re = re.compile(r"\d+\.?\d*")  # регулярное выражение для чисел
        self.fit_words(train_sents)
        self.fit_tags(train_sents)
        X, y = self.build_X_y(train_sents)
        self.fit_condition_prob_matrix(X, y)
        self.fit_prob_matrix(X, y, coeff=coeff)
        self.fit_start_probs(train_sents)

    def word_prepare(word, first_in_sentense):
        """Приводит слово к нейтральному виду, в котором
        оно хранится в словаре."""
        upper_case = word.istitle()
        word = word.lower()
        re_res = HMMTagger.number_re.match(word)
        if (not re_res is None) and (re_res.span() == (0, len(word))):
            # print(word)
            return "17"  # заменяем все числа на фиксированное число.
                         # это увеличивает accuracy на test_sent[:50]
                         # на 1.2% и уменьшает словарь на ~ 1300 слов
        if (not first_in_sentense) and (upper_case):
            return "Julia"  # если слово в середине предложения написано
                            # с большой буквы - это имя собственное
                            # заменим все обнаруженные на какое-то одно
        return word

    def fit_words(self, train_sents):
        """Составляет словарь слов по выборке."""
        self.words = dict()
        for sent in train_sents:
            for i, (word, tag) in enumerate(sent):
                word = HMMTagger.word_prepare(word, i==0)
                # конечно, confidence и Confidence - одно слово
                total = len(self.words)
```

```
            if not (word in self.words):
                self.words[word] = total

    def fit_tags(self, train_sents):
        """Составляет прямой и обратный словари тегов."""
        self.tags = dict()
        self.reverse_tags = []
        for sent in train_sents:
            for word, tag in sent:
                total = len(self.tags)
                if not (tag in self.tags):
                    self.tags[tag] = total
                    self.reverse_tags.append(tag)

    def encode_tag_pair(self, t1, t2):
        """По элементам пары вычисляет её код."""
        return t1 * len(self.tags) + t2

    def decode_tag_pair(self, code):
        """По коду пары возвращает её элементы."""
        return int(code) // len(self.tags), int(code) % len(self.tags)

    def build_X_y(self, train_sents, sep = "."):
        """Построение выборки в явном (X, y) виде."""
        # sep - разделитель предложений
        sep_id = self.tags[sep]

        y = []
        for sent in train_sents:
            for i, (word, tag) in enumerate(sent):
                y.append(self.words[HMMTagger.word_prepare(word, i==0)])
            # y.append(self.words[sep])


        X = []
        last_tag = sep_id  # точка - это тег
        for sent in train_sents:
            for word, tag in sent:
                tag_id = self.tags[tag]
                X.append(self.encode_tag_pair(last_tag, tag_id))
                last_tag = tag_id
            # X.append(self.encode_tag_pair(last_tag, sep_id))
```

```
            # last_tag = sep_id

        return X, y

    def fit_condition_prob_matrix(self, X, y):
        """Вычисление условных вероятностей."""
        self._unknown_word_id = len(self.words)
        self.condition_prob_matrix = np.zeros((len(self.words) + 1, len(self.tags)))
        # матрица переходов
        # все пары ненулевых элементов prob_matrix, каждая по одному разу
        # последняя строчка для неизвестного случайного слова

        for i in range(len(X)):
            t1, t2 = self.decode_tag_pair(X[i])
            self.condition_prob_matrix[y[i], t2] += 1  # считаем сумму I(Yi =w, Xi = t)

        self.condition_prob_matrix[-1, :] = (
            self.condition_prob_matrix[:-1, :].mean(axis=0)
        )
        # для неизвестных слов. В качестве альтернативы можно было заменять неизвестное
        # слово на известное, например, this - 0.932716568545
        # проставить равномерные условные вер-ти -  0.942809083263
        # сделать, как выше - 0.956265769554
        # (оценки accuracy по test_sents[:50] - выше говорилось,
        # почему для этого стоило отвести отдельную подвыборку)

        cnt_xi = self.condition_prob_matrix.sum(axis = 0)  # сумма I(Xi = t)

        for y_value in range(len(self.words) + 1):
            for tag in range(len(self.tags)):
                self.condition_prob_matrix[y_value, tag] /= cnt_xi[tag]

        assert ((np.abs(self.condition_prob_matrix.sum(axis=0) -
                    np.ones(len(self.tags))) < 1e-9).sum() == len(self.tags))
        # проверяем, что получились одни единицы

    def fit_prob_matrix(self, X, y, coeff=1, log=False):
        """Вычислени матрицы переходов для X, y.
        Значение coeff подобрано экспериментально"""
        states_count = len(self.tags) * len(self.tags)
        nij = np.zeros((states_count, states_count), dtype=float)
        # nij = sum_k I(X_{k-1} =i, X_k = j)
```

```
ni = np.zeros(states_count, dtype=int)
#ni = sum_j nij

for i in range(len(X) - 1):  # собственно, считаем вхождения пар
    nij[X[i], X[i + 1]] += 1

##################
### Для того, чтобы все тройки были возможны, добавим константу к
### разрешённым переходам в матрице, иначе - некоторые тройки станут
### недопустимыми и на новых предложениях будет плохой прогноз.
### !!! Ниже будет обоснован выбор константы !!!
for t1 in range(len(self.tags)):
    for t2 in range(len(self.tags)):
        for t3 in range(len(self.tags)):
            x1 = self.encode_tag_pair(t1, t2)
            x2 = self.encode_tag_pair(t2, t3)
            nij[x1][x2] += coeff
##################

ni = nij.sum(axis = 1)

self.prob_matrix = np.zeros(
    (len(self.tags), len(self.tags), len(self.tags))
)
# prob_matrix[t1, t2, t3] = P(X_{i+1} = (t2, t3)| X_i = (t1, t2))

for t1 in range(len(self.tags)):
    for t2 in range(len(self.tags)):
        i = self.encode_tag_pair(t1, t2)
        for t3 in range(len(self.tags)):
            j = self.encode_tag_pair(t2, t3)
            self.prob_matrix[t1, t2, t3] = (
                nij[i, j] / ni[i] if nij[i, j] > 0 else 0
            )

if log:
    print(self.prob_matrix.sum(axis = 2))
    # должны быть 1 для достижимых состояний и 0 для недостижимых
    # (т.е. таких пар, которых не встретилось в обучающей выборке)

# проверим, что корректно подсчиталось
```

```
        for line in self.prob_matrix.sum(axis = 2):
            for val in line:
                assert (abs(val -1.) < 1e-9) or (abs(val - 0.) < 1e-9)

        self.log_prob_matrix = np.log(self.prob_matrix)
        # один вызов для всей матрицы раз в пять ускоряет работу,
        # в сравнении с множеством вызовов для одного числа

    def fit_start_probs(self, train_sents):
        """Вычисление начального распределения."""
        self.start_probs =  np.zeros(len(self.tags))
        for sent in train_sents:
            self.start_probs[self.tags[sent[0][1]]] += 1
        self.start_probs += 1. / len(train_sents)
        self.start_probs /= self.start_probs.sum()

    def my_log(f):
        """Вычисление логарифма с условием log(0) = - inf"""
        if f != 0:
            return np.log(f)
        else:
            return 0. - np.inf

    def eval_viterbi(self, y, log=False):
        """Вычисляет траекторию Витерби для Y=y"""
        T = len(y)

        g1 = np.zeros((len(self.tags), len(self.tags)))
        # g1 из презентации для вершины x = (t1, t2)

        for t1 in range(len(self.tags)):
            for t2 in range(len(self.tags)):
                g1[t1, t2] = (HMMTagger.my_log(self.start_probs[t2]) +
                              HMMTagger.my_log(self.condition_prob_matrix[y[0], t2]))


        g = np.zeros((T, len(self.tags), len(self.tags), len(self.tags)))
        # g из презентации, где g[T, t1, t2, t3] соответствует переходу из
        # состояния x1 = (t1, t2) в состояние x2 = (t2, t3) во время T (y[T])
        # это нужно, т.к. переходы из (a, b) в (c,d), где b != c невозможны,
        # а хранение лишней памяти - растратно и медленно (промахи кэша и выделение памяти)
```

```
        """
        for t1 in range(len(tags)):
            for t2 in range(len(tags)):
                for t3 in range(len(tags)):
                    g[0, t1, t2, t3] = g1[t2, t3]
        """
        # Код снизу делает то же самое, что и код сверху
        for t1 in range(len(self.tags)):
            g[0, t1, :, :] = g1


        for t in range(T):
            if (log):
                print(t, end=",")
            for t3 in range(len(self.tags)):
                v2 = HMMTagger.my_log(self.condition_prob_matrix[y[t], t3])
                g[t, :, :, t3] = self.log_prob_matrix[:, :, t3] + v2


        ################

        G_dp_vals = np.zeros((T, len(self.tags), len(self.tags)))   # G* в презентации
        # состояния динамики кодируем так же - G_dp_vals[T, t1, t2]
        # соответствует G_dp[T, x], где x = (t1, t2)
        # - уменьшим расход памяти в len(tags) раз.

        G_dp_ways = np.zeros((T, len(self.tags), len(self.tags)), dtype=int)
        # будем запонимать, откуда пришли, чтобы восстанавливать ответ проще.

        # инициализация ДП
        G_dp_vals[0, :, :] = g1[:, :]
        G_dp_ways[0, :, :] = -1
        # храним пару - значение и номер ячейки, откуда пришли

        for t in range(1, T):
            if (log):
                print(t, end=",")

            for t3 in range(len(self.tags)):
                possible_values = (G_dp_vals[t - 1, :, :] + g[t, :, :, t3])
                # на месте первого двоеточия - индекс, который мы
                # перебираем, максимизируя результат
```

```
                argmax = (G_dp_vals[t - 1, :, :] + g[t, :, :, t3]).argmax(axis=0)

                G_dp_vals[t, :, t3] = possible_values[argmax, range(len(self.tags))]
                # тоже самое, что и
                # G_dp_vals[t, t2, t3] = possible_values[argmax[t2], t2] в цикле по t2
                G_dp_ways[t, :, t3] = argmax[:]
                # вычислим максимум с помощью numpy, это быстрее

        ################


        argmax = G_dp_vals[T - 1, :, :].argmax()
        k_star = (argmax // len(self.tags), argmax % len(self.tags))

        if (log):
            print((G_dp_vals[T-1, :, :] != -np.inf).sum())

        ans = [-1] * T
        for t in range(T)[::-1]:
            ans[t] = (y[t], self.reverse_tags[k_star[1]])
            previous_tag = G_dp_ways[t, k_star[0], k_star[1]]
            k_star = (previous_tag, k_star[0])

        return ans

    def tag(self, test_sent, log=False):
        """Предсказывает вектор тегов.
        Входные данные - список слов в предложении.

        Возвращает  список пар (слово, тег)
        """

        y = []
        if log:
            print("Test sentenses = ", test_sent)
        for i, word in enumerate(test_sent):
            word_prep = HMMTagger.word_prepare(word, i==0)
            if word_prep in self.words:
                y.append(self.words[word_prep])
            else:
                if log:
                    print("Unknown word:", word)
```

```
                y.append(self._unknown_word_id)

        return self.eval_viterbi(y)
```

Ещё раз запустим код (уже класса) на 50 первых предложениях из тестовой выборки

In [149]:
```
def score(model, test_sents):
    tagging_test = [model.tag([word for word, tag in sent]) for sent in test_sents]
    correct_tags = [tag for sent in test_sents for word, tag in sent]
    predict_tags = [tag for sent in tagging_test for word, tag in sent]
    # print(len(correct_tags), correct_tags[:20])
    # print(len(predict_tags) , predict_tags[:20])
    score = 0.
    for t1, t2 in zip(correct_tags, predict_tags):
        if (t2 == None):
            t2 = "NN"
        if (t1 == t2):
            score += 1
    score /= len(correct_tags)
    return score
```

In [150]:
```
%%time
hmm_tagger = HMMTagger(train_sents)
print("tagger = ", hmm_tagger)
print("accuracy = ", score(hmm_tagger, test_sents[:50]))
```

```
tagger =  <__main__.HMMTagger object at 0x7f06dcc672b0>
accuracy =  0.9571068124474348
CPU times: user 15.2 s, sys: 116 ms, total: 15.3 s
Wall time: 15.4 s
```

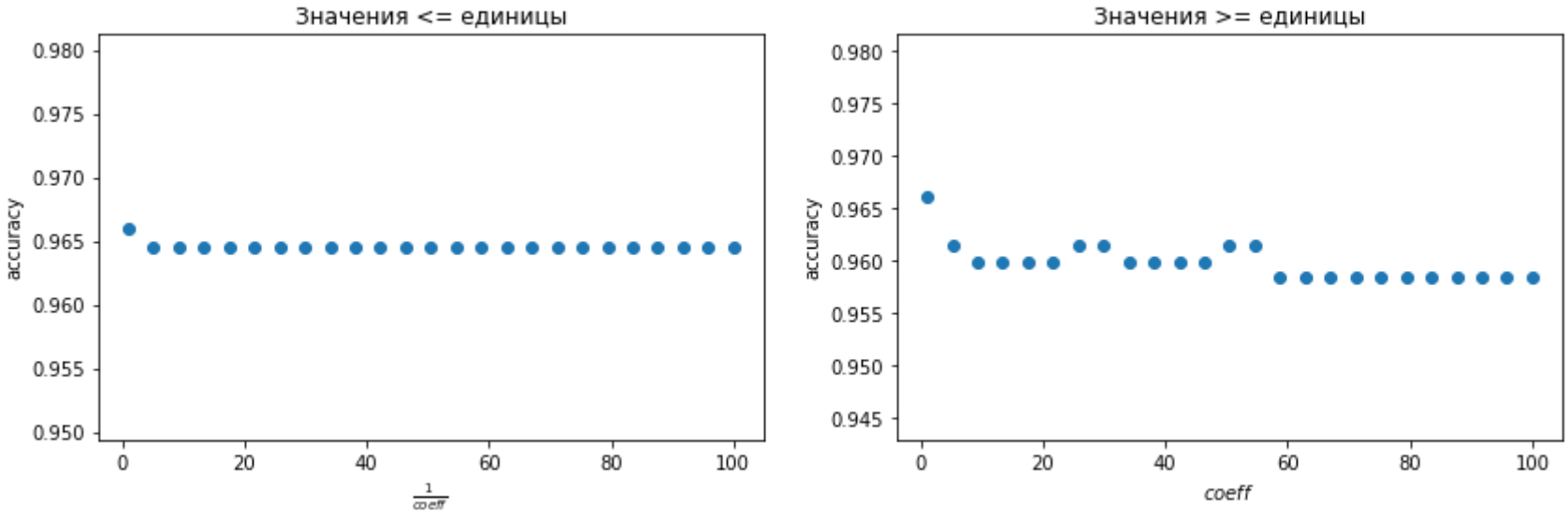
Получили хорошее значение. Теперь обоснуем выбор константы. Для этого, с помощью **поиска по сетке** найдём такое значение константы, что accuracy будет наибольшим. Вычисления, разумеется, будем производить на обучающей выборке (иначе результаты подсчёта accuracy будут некорректы и контролировать преобучение будет невозможно. Из-за малой скорости работы, будем считать accuracy не по всей выборке, а по sent_run псевдослучайно выбранных предложений (чтобы не нарваться на закономерности в данных) Для этого выберем предложения с номерами, кратными len(train_sents) // sent_run (можно было бы и случайно). При этом, половина значений пусть будет <= единицы (1/np.linspace), половина - >= (np.linspace)

In [73]:
```
%%time

sent_run = 30
log = False
params = list(1. / np.linspace(1., 100, 25)) + list(np.linspace(1., 100, 25))
ret = []

sub_sents = [train_sents[8000//sent_run*i] for i in range(sent_run)]
for param in params:
    tagger = HMMTagger(train_sents, coeff=param)
    fit_prob_matrix(coeff=param)
    total_accuracy = score(tagger, sub_sents)
    if log:
        print("Total accuracy = ", total_accuracy)
    ret.append((total_accuracy, param))

if log:
    print(ret)
```

```
CPU times: user 9min 57s, sys: 5.24 s, total: 10min 3s
Wall time: 10min 3s
```

In [74]:
```
ret = np.array(ret)
import pickle
with open("ret_set_nij_param.pkl", "wb") as f:
    pickle.dump(ret, f)

with open("ret_set_nij_param.pkl", "rb") as f:
    check_ret_save = pickle.load(f)

print("Saved = ", (check_ret_save == ret).sum() == ret.shape[0] * ret.shape[1])

plt.figure(figsize=(14, 4))
plt.subplot(121)
plt.scatter(1./ret[:len(ret)//2, 1],  ret[:len(ret)//2, 0])
plt.xlabel("$\\frac{1}{coeff}$")
plt.ylabel("accuracy")
plt.title("Значения <= единицы")
plt.subplot(122)
plt.scatter(ret[len(ret)//2:, 1],  ret[len(ret)//2:, 0])
plt.xlabel("$coeff$")
plt.ylabel("accuracy")

plt.title("Значения >= единицы")
plt.show()

best = ret[ret[:, 0].argmax(), :]
# print(ret)
print("Максимальное accuracy = %f при param = %f" % tuple(best))
```

```
Saved =  True
```




```
Максимальное accuracy = 0.966102 при param = 1.000000
```

Найдено оптимальное значение параметра для данной крупности сетки.

---

## Запуск на всей выборке

Сравним accuracy с предложенными моделями:

In [111]:
```
%%time
hmm_tagger = HMMTagger(train_sents, coeff=1.)  # 0.9480562519812109
unigram_tagger = nltk.UnigramTagger(train_sents)
bigram_tagger = nltk.BigramTagger(train_sents)
combined_bigram_tagger = nltk.BigramTagger(train_sents, backoff=unigram_tagger)
```

```
CPU times: user 17.5 s, sys: 176 ms, total: 17.7 s
Wall time: 17.7 s
```

In [112]:
```
%%time
for tagger in [hmm_tagger, unigram_tagger, bigram_tagger, combined_bigram_tagger]:
    print("tagger = ", tagger)
    print("accuracy = ", score(tagger, test_sents))
```

```
tagger =  <__main__.HMMTagger object at 0x7f06de1d1240>
accuracy =  0.9480562519812109
tagger =  <UnigramTagger: size=18164>
accuracy =  0.9072072850926486
tagger =  <BigramTagger: size=44203>
accuracy =  0.32501945188899456
tagger =  <BigramTagger: size=2953>
accuracy =  0.9188207832627302
CPU times: user 6min 1s, sys: 456 ms, total: 6min 1s
Wall time: 6min 2s
```

**Вывод:** Мы реализовали крутой (**94.8% accuracy**) алгоритм, который умеет анализировать связи слов в предложении и лучше трёх представленных для сравнения алгоритмов (91.88% - BigramTagger), в том числе "Простейшего POS-tagging'a" (90.7%). Из его недостатков - сравнительно медленная работа (**6 минут на test_sents**), хотя конктретная данная реализация значительно оптимизирована (42 раза по времени в сравнении с реализацией без множественных операций numpy, в 44 раза меньше памяти за счёт того, что храним урезанные таблицы переходов и проч.)